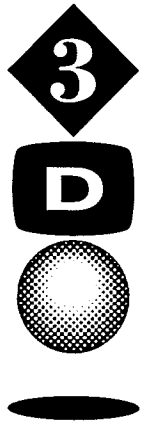


3 D O M 2 R E L E A S E ♦ V E R S I O N 2 . 0

Volume 3

- ♦ *3DO M2 Tools for Sound Design*
- ♦ *3DO M2 Audio and Music Programmer's Guide*
- ♦ *3DO M2 Audio Programmer's Reference*



3DO M2 Tools for Sound Design

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

3DO and the 3DO Logo are trademarks and/or registered trademarks of The 3DO Company. ©1996 The 3DO Company. All rights reserved. Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

About This Document.....	TSD-xiii
Audience.....	TSD-xiii
How This Document Is Organized	TSD-xiii
Typographical Conventions	TSD-xiv

1

Music and Sound Effects for 3DO Titles

Introduction.....	TSD-1
Chapter Overview	TSD-1
Resource Allocation and Planning	TSD-2
RAM-Resident vs. ROM-Resident Sound	TSD-2
Spooled vs. Streamed Sound	TSD-2
Sound File Characteristics	TSD-3
Audio Production.....	TSD-4
Sampling Sound Effects	TSD-4
Working With Sampled Music	TSD-5
Creating Algorithmic Sound Effects	TSD-6
Incorporating Sound Effects Into Your Title	TSD-6
Playing MIDI Scores	TSD-6
Streaming Audio Data With the 3DO DataStreamer	TSD-7
Aesthetic Considerations	TSD-7
Tweak, Tweak, Tweak!	TSD-7
Project Management	TSD-8

2

Installing and Launching ARIA

Introduction	TSD-11
Chapter Overview	TSD-11
Hardware and Software Configuration	TSD-12
Launching and Setting Up ARIA	TSD-12
Troubleshooting	TSD-13
Setup for Working With MIDI	TSD-13
Files and Folder Setup	TSD-13
Required Files for Real-time MIDI	TSD-13
Setting Up for MIDI	TSD-14
Drag and Drop ARIA Installation	TSD-14

3

Creating MIDI Projects With ARIA

Introduction	TSD-17
Chapter Overview	TSD-17
Preparing Your Environment	TSD-18
MIDI Files and PIMap Files	TSD-18
Working With PIMaps	TSD-18
Playing a MIDI File on the 3DO Station	TSD-20
Working With General MIDI	TSD-22
Working With Real-time MIDI	TSD-23
Tips and Tricks	TSD-23
MIDI Project Interface Tips	TSD-23
MIDI Tricks	TSD-24

4

SquashSnd

Introduction	TSD-25
SquashSnd-Compatible File Types	TSD-25
SquashSnd Parameters	TSD-26
Caveats	TSD-27

5

3SF Overview

Introduction	TSD-29
The 3DO Score File Format	TSD-30
3SF Architecture	TSD-30

Before 3SF	TSD-30
Method 1: Play a Sample in Memory	TSD-31
Method 2: Spool a Sample From Disk	TSD-31
Method 3: Weave a Sample Into a Stream	TSD-31
Method 4: Play a Synthetic Patch in the DSP	TSD-31
Method 5: Play a MIDI File	TSD-31
Sound Design With 3SF	TSD-32
Score File Creation	TSD-32
From ARIA	TSD-32
MakeScore Tool	TSD-33
Score File Playbackscore	TSD-33
From ARIA	TSD-33
PlayScore	TSD-33
From a C or C++ Program	TSD-33
.....	TSD-33

6

3SF Architecture

Overview	TSD-35
Creating a 3DO Score File	TSD-36
Playing a 3DO Score File	TSD-37

7

3SF User's Guide

Introduction	TSD-39
Why Use 3SF	TSD-40
The 3SF Tool Set	TSD-40
MakeScore	TSD-41
DumpIFF	TSD-42
PlayScore	TSD-43
PlayScore.lib	TSD-43
ScorePlayer	TSD-43
ErrorProc	TSD-44
StartReadingAndPlaying	TSD-44
StartReading	TSD-44
Play, Pause, Stop	TSD-45

8

Essential Features of SoundHack

Introduction	TSD-47
Chapter Overview	TSD-47
Installing SoundHack.....	TSD-48
Converting a Sound File to Compressed Format.....	TSD-48
Loading a Sound File	TSD-48
Playing the Current Sound File	TSD-49
Converting the Current Sound File to Compressed Format	TSD-49
Reading/Writing Raw Files.....	TSD-49
Changing Header Information.....	TSD-50
Header Change Command	TSD-50
Loops & Markers Command	TSD-51

9

SoundHack User's Guide

Introduction.....	TSD-53
Chapter Overview	TSD-53
The File Menu	TSD-54
Open Command	TSD-54
Open Any Command	TSD-55
Close Command	TSD-55
Save A Copy Command	TSD-56
lisTen to AIFF File command	TSD-56
Import SND Resource Command	TSD-56
Export SND Resource Command	TSD-56
Quit Command	TSD-56
The Edit Menu.....	TSD-56
The Hack Menu.....	TSD-56
Binaural Filter	TSD-57
Processing a File With the Binaural Filter	TSD-58
Convolution Command	TSD-58
Gain Change Command	TSD-61
Mutation Command	TSD-61
Phase Vocoder Command	TSD-64
Spectral Dynamics Command	TSD-66
Varispeed Command	TSD-68
The Control Menu	TSD-70
Show Output Command	TSD-70

Show Spectrum Command	TSD-70
Pause Process Command	TSD-70
Continue Process Command	TSD-70
Stop Process Command	TSD-70
Bibliography	TSD-71

10

AudioThing

Introduction	TSD-73
System Requirements and Options	TSD-73
Installation	TSD-74
Using AudioThing	TSD-74

11

Tips, Tricks, and Troubleshooting

Available audio tools	TSD-75
SoundHack	TSD-75
SquashSound	TSD-75
ARIA	TSD-75
MakePatch	TSD-76
AIFF Files	TSD-76
AIFF files in Examples folder	TSD-76
AIFF and compression	TSD-76
Preparing Audio	TSD-76
Mastering audio	TSD-76
Editing Synthesized Audio	TSD-76

A

AIFF Samples

Overview	TSD-79
File names	TSD-80
Auditioning samples from the Sound Designer II Edit window	TSD-80
Folder contents	TSD-81
Samples in GMPercussion22k and GMPercussion44k	TSD-81
Samples in the PitchedL folder	TSD-83
Samples in the PitchedLR folder	TSD-83
Samples in the Unpitched folder	TSD-83

B

DumpIFF Sample Output

Introduction	TSD-87
--------------------	--------

C

3SF File Format Specification

Introduction	TSD-89
File Types and Extensions	TSD-89
Data Types	TSD-90
File Structure	TSD-92
Chunk	TSD-92
FORM Chunk	TSD-93
Syntax Definitions	TSD-94
File Version Chunk	TSD-95
Sound Set	TSD-95
Score Items	TSD-96
Licks	TSD-97
Axes	TSD-97
Item Numbers and Item Names	TSD-97
Set Header	TSD-99
Item Header	TSD-99
The Global Numeric Index	TSD-100
Specifying a Score Item or a Sound Set	TSD-101
The Name Index	TSD-102
Dependency Lists	TSD-104
MIDI Context Format	TSD-104
MIDI Event Format	TSD-105

List of Figures

Figure 2-1 Communication Preferences dialog.....	TSD-12
Figure 2-2 PatchBay setup for working with ARIA.....	TSD-14
Figure 3-1 MIDI project with PIMap file.....	TSD-21
Figure 3-2 MIDI project window with 3DO MIDI Player dialog box.....	TSD-22
Figure 6-1 Creation of a 3SF file.	TSD-36
Figure 6-2 Three ways to play back a 3SF file.....	TSD-37
Figure 8-1 Soundfile Information window.	TSD-48
Figure 8-2 Output Soundfile Format dialog for changing to Headerless Data file type.	TSD-50
Figure 8-3 Header change dialog.	TSD-50
Figure 8-4 Loops and Markers dialog.	TSD-51
Figure 9-1 File menu.....	TSD-54
Figure 9-2 Soundfile Information dialog.	TSD-55
Figure 9-3 Hack menu.	TSD-57
Figure 9-4 Binaural Position Filter dialog.	TSD-57
Figure 9-5 Convolution dialog.	TSD-59
Figure 9-6 Example for moving convolution.....	TSD-60
Figure 9-7 Gain Change dialog.	TSD-61
Figure 9-8 Spectral Mutation dialog.	TSD-62
Figure 9-9 Phase Vocoder dialog.....	TSD-64
Figure 9-10 Edit Function dialog.....	TSD-66
Figure 9-11 Spectral Dynamics Processor dialog.....	TSD-67
Figure 9-12 Varispeed dialog.....	TSD-69
Figure 9-13 Varispeed Edit Function dialog.	TSD-69
Figure 9-14 Control menu.	TSD-70
Figure M-1 Basic layout of a chunk.	TSD-92
Figure M-2 Structure of the main 3SF FORM chunk.	TSD-94

List of Tables

Table 3-1	Flags in a PIMap	TSD-19
Table 9-1	Convolution dialog options.	TSD-59
Table 9-2	Spectral Mutation Function dialog options.	TSD-62
Table 9-3	Phase Vocoder dialog options.	TSD-64
Table 9-4	Spectral Dynamics Processor dialog options.	TSD-67
Table 9-5	Varispeed dialog options.	TSD-69

Preface

About This Document

This document gives an introduction to music and sound effects for 3DO™ titles, provides tutorial-style and reference information for all 3DO Sound Designer tools, and contains additional troubleshooting and reference information.

Audience

This book is for sound designers who want to prepare music for the 3DO Station. While some background in composing digital audio is useful, it is not required. To understand how to work with the MIDI synthesizing capability of the ARIA tool, you need at least a rudimentary background in MIDI.

How This Document Is Organized

- ◆ Chapter 1, "Music and Sound Effects for 3DO Titles," was written by a composer working for The 3DO Company™. It gets you started with 3DO audio.
- ◆ Chapter 2, "Installing and Launching ARIA," describes launching and setting up ARIA as well as how to use it with MIDI.
- ◆ Chapter 3, "Creating MIDI Projects With ARIA" describes how to use the PiMAP part of ARIA.
- ◆ Chapter 4, "SquashSnd," explains the use of the SquashSnd MPW tool for compressing and decompressing audio files.
- ◆ Chapter 5, "3SF Overview," introduces the all-in-one 3DO Sound File Format.
- ◆ Chapter 6, "3SF Architecture," discusses 3SF's supporting software architecture.
- ◆ Chapter 7, "3SF User's Guide," tells you how to use the 3SF format in sound design.

- ◆ Chapter 8, “Essential Features of SoundHack,” provides a tutorial-style introduction to all major features of the SoundHack tool.
- ◆ Chapter 9, “SoundHack User’s Guide,” provides a detailed reference to all commands and dialog options.
- ◆ Chapter 10, “AudioThing,” describes a sound-decompression tool that lets you preview what audio files will sound like when played on 3DO hardware.
- ◆ Chapter 11, “Tips, Tricks, and Troubleshooting,” provides troubleshooting information, mostly based on questions asked by developers and answered by 3DO technical staff.
- ◆ Appendix A, “AIFF Samples,” lists all AIFF samples on the Toolkit CD-ROM. These samples are useful for experimenting with the tools described in this document, and also may be used in a 3DO title.
- ◆ Appendix B, “DumpIFF Sample Output,” contains a 3SF file dump.
- ◆ Appendix C, “3SF File Format Specification,” describes 3SF data types and file structures. Keep in mind that this information is provisional, pro tem, and in flux.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>Scene_GetStatic(scene)</code>
procedure name	<code>Char_TotalTransform()</code>
new term or emphasis	In M2, <i>characters</i> are objects that can be displayed on the screen.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.

Music and Sound Effects for 3DO Titles

Introduction

The 3DO system makes it possible to use CD-quality stereo in a title. Preparing music and sound effects therefore requires special care. This chapter helps in planning and preparing music and sound effects for a 3DO title.

Chapter Overview

This chapter discusses the following topics:

Topic	Page
Resource Allocation and Planning	2
Audio Production	4
Incorporating Sound Effects Into Your Title	6
Aesthetic Considerations	7
Project Management	8

Resource Allocation and Planning

Before starting audio production, it's helpful to decide how a title will use audio and where the audio files will reside. This section discusses the choices available. Note that considerations that apply to MIDI files also apply to 3SF files.

- ◆ Audio types:
 - ◆ Samples (AIFF or raw files)
 - ◆ DSP instruments
 - ◆ MIDI (using multiple sample files or DSP instruments)
 - ◆ 3SF files (all-in-one file containing any of the above)
- ◆ Run-time location of files: RAM-resident or CD-ROM-resident.
- ◆ Playback for CD-ROM-resident sounds: spooled or streamed from disc.
- ◆ RAM budget: stereo or mono, sample rate, and/or 2:1 compression.
- ◆ CD bandwidth budget, in K/sec.

RAM-Resident vs. ROM-Resident Sound

First decide whether the title itself will be RAM-resident or will use data streaming.

- ◆ A RAM-resident title places all necessary code within the 4 MB—or actually 3.2 MB—of the available system RAM during any run-time segment. RAM-resident titles leave the CD-ROM available for accessing audio through spooling, streaming, or loading.
- ◆ Titles stream most data and code on a real-time, interactive basis from the CD-ROM. In this case, you cannot spool an audio sample file, but you *can* include audio as part of the data stream.

Since seek times—the time it takes to find the sound on the CD-ROM—limit how interactive sound can be, RAM-resident sounds have a distinct advantage. It is possible, however, to have multiple audio streams and dynamically fade, pan, and mix them according to what's going on in the title.

Spooled vs. Streamed Sound

Spooled sound comes directly from a file on disc. A program can also play short individual sound effects like an explosion or a door opening while spooling. Background music can be spooled in using a separate task.

Streamed sound is played using functions from the 3DO DataStreamer library. The advantage of working with streamed audio is that it allows a title to benefit from audio sample files, even though the title is not RAM-resident. Note that you can't stream more than 300 KB/sec total of video and audio data.

For More Information

- ◆ The *3DO Jumpstart for Programmers* provides examples for playing a sound effect and for spooling sound with the Sound Spooler.
- ◆ The *3DO DataStreamer Programmer's Guide* discusses the DataStreamer.

Sound File Characteristics

Decisions about where sound will reside and how it will be played influence decisions about the characteristics of the sound file you generate. There's always a trade-off between file size and sound quality.

Stereo vs. Mono

An early consideration should be whether to use stereo or mono sounds. Remember that stereo sound at a 44.1 kHz sample rate requires 10 MB per minute. If mono is adequate, the size is cut in half. Certain sound effects require stereo (panning, flanging, tennis matches, etc.), but many sound effects can be mono. If CD-ROM space is available, spooled or streamed files should be stereo, since more and more television sets provide stereo sound.

Sample Rate

When you decide on the sample rate, remember that many televisions produce high-quality sound. You may not be able to get by with 22 kHz samples. Take care to avoid audio aliasing: Sample any given sound at twice the frequency of its highest component. Human speech and other sounds with many upper partials may sound boxy at 22 kHz.

Compression

You can use compressed sound as long as the correct DSP instrument is attached during playback.

- ◆ For 2:1 compression, use the `squashsnd` MPW tool, which is part of the 3DO system release, or the SoundHack tool, which is documented in this book.
- ◆ For 4:1 ADPCM (Adaptive Differential Pulse Code Modulation) compression, you can also use SquashSnd (and MPW tool) or SoundHack. At 3DO, developers have found, however, that a 22 kHz sound compressed at a 2:1 ratio is less distorted than a 44.1 kHz sound compressed at a 4:1 ratio.

When you decide whether to compress your audio files, remember that there is a DSP bandwidth trade-off: Compression saves RAM/ROM but decompression of most formats can result in a big system resource overhead hit because the DSP code required to decompress sample files takes up more DSP cycles and code space.

Note: *The mono SQS2 format is decompressed in hardware, and has less additional overhead, compared to 16-bit data.*

The M2 Operating System, however, has more DSP code space and better DSP performance than the Portfolio Operating System. Because of these improvements, decompression should not have as great an impact on system resources, unless you have many sound effects operating at the same time.

M2 also has a new 2:1 compression that requires minimal DSP usage.

Audio Production

Use the best recording environment available for title development, whether you are sampling a small sound effect or producing a live orchestral score.

Sampling Sound Effects

You can use sound effects from the 3DO Content Library or record the effects yourself.

- ◆ The Content Library provides almost 20,000 copyright-free sound effects (and 60 hours of music), which you can access using the 3DO CD Browser. The Content Library is available for the cost of production—contact The 3DO Company for more details.
- ◆ If you need only a few sound effects, consider recording them yourself, or use MakePatch to synthesize them.

Preparing AIFF Files

A variety of tools are available for working with AIFF files:

- ◆ To convert sound effects from Red Book format to AIFF format, use the Disc-to-Disk tool from Optical Media International.
- ◆ To edit AIFF files, use Digidesign's Sound Designer II. Note that Sound Designer employs a software filter in its edit window. To audition 22 kHz or lower SR files, click on the Play tool in the dialog box that appears when you choose Open from the File menu.

Caution: *Having the AudioMedia board's inputs and outputs looped through a mixer produces a painfully loud sound when you go into the record window of Sound Designer. Lower the output faders when recording.*

- ◆ To convert audio file formats and sample rates, use SoundHack.
 - ◆ From the Hack menu, choose Varispeed for SR conversions.

- ◆ From the File menu, choose Save A Copy, then select the file type Audio IFC and the Format 3DO 2:1 SDXC (or the 4:1 compression if you prefer).
- ◆ For 2:1 and ADPCM compression, use the MPW tool SquashSnd. This is especially helpful for batch-processing files.

Testing Audio/Video Synchronization

To test synchronization of a sound effect with an animation or a short video clip, you can use 3DO Animator:

1. Save the sound as Macintosh System 7 Sound Resource (using Sound Designer II, 3DO SoundHack, or SoundEditPro).
2. In 3DO Animator, load the animation and move it to the desired starting frame, then click the Sound button in the Anim Control Panel and select the sound resource file you want to attach.
3. Make sure to use the same frame rate as the title will use (or record the frame you are happy with and convince the rest of the team to go with that rate). Then play the animation using the Play button (single arrow) on the control panel.
4. If you want to save the audio attachment, save the file in VDAN format.
5. Remember that the final animation has to be in 3DO file format and the sound file has to be in AIFF file format.

Working With Sampled Music

If you want to include music in a title, record in the best studio environment available or consider production music from the Content Library. The music in the Content Library includes pieces indexed by composer, by instrument, and by type—for example, “sad” or “exuberant.”

Note: *If you use music from the 3DO Content Library, remember that there is no guarantee that other developers have not selected the same composition for their title.*

If you create your own music, you can produce an AIFF file using a hard disk recording system or DAT. Then perform a digital transfer to Sound Designer II or some other sample editing program for final edits—for example, to make sure a sample is as small as possible, to do SR conversions, to find loop points.

Since music files need memory, you and the producer have to decide on ROM and RAM budgets for your files. Looping music or sound effects files—for example, the sound of fire—may reduce file size, but may compromise drama and realism. To make the loop point less noticeable, you can use tricks like a vague end or a

“timed end,” where some rhythmic oscillation between sound and silence covers seek time. You need to experiment with this since different audio buffer sizes change the length of silence at the loop point.

Testing Synchronization

To test synchronization of a music file you can use Adobe Premiere. For other files, use 3DO Animator. Note that Adobe Premiere does *not* read or write SMPTE (Society of Motion Picture and Television engineers; think of the SMPTE standard as a conductor for synchronizing separately recorded audio and video tape), but it facilitates editing QuickTime movies *and* it can mix its three digital audio tracks when you save a videoless movie.

If SMPTE is available in your development environment, you can lock to video and set audio sync on tape. Then transfer this sync to the 3DO system by coordinating SMPTE numbers to your animation. Sound Designer II allows you to stipulate horizontal measurements in SMPTE. Premier or Passport’s Producer can synchronize a QuickTime movie from an animation to SMPTE frame numbers.

Creating Algorithmic Sound Effects

The 3DO system lets you change existing sounds programmatically by adding filters, changing the pitch, or adding an envelope. Use MakePatch, or wWork with programs to achieve program-controlled sound effects, for example, engines based on samples, or purely algorithmic sound effects based on synthetic sounds.

Incorporating Sound Effects Into Your Title

Once you’re satisfied with the sound effects or music you have produced, a programmer has to incorporate it into the title in the appropriate way. This section explains how sound can be played on the 3DO system. It helps nonprogrammers understand the principles behind audio on the 3DO system and gives references to the appropriate sections of the *3DO Portfolio Programmer’s Guide* for more in-depth coverage.

Playing MIDI Scores

The 3DO system is set up to allow MIDI score playing, which is more interactive than spooling. You can use the MakePatch tool to develop synthetic patches (instruments) or to attach sample files to MIDI notes.

You can attach MIDI notes to synthetic sounds or AIFF samples, depending on RAM availability and system resources. Synthetic instruments take the least amount of RAM, but may result in a small processing overhead. On the Release 2.0 software CD-ROM, you can find a library of AIFF samples to use in your titles. An overview is provided in Appendix A, “AIFF Samples.”

You can use hierarchical score structures for more dramatic sound design; for more information, see the documentation on the Music Library.

Streaming Audio Data With the 3DO DataStreamer

You can fade or mix several channels of digital audio samples for dramatic purposes, but remember bandwidth limitations. The *3DO DataStreamer Programmer's Guide* discusses data streaming and provides manpages for all relevant functions in the DataStreamer library.

Aesthetic Considerations

If your title development team has experience only with traditional cartridge games or games running on a computer, consider using Hollywood or other professional music talent to make Silicon Valley titles rock, hop, sing, and thrill. The 3DO environment's CD-quality audio capability lets developers take advantage of experience from the "real music world." Big names will also help a product's market appeal, but project management may be more difficult because of the tight schedules of well-known composers or producers.

Interactive titles allow for a multiplicity of musical styles, both to distinguish one game from all others and within one game. For example, you can associate different musical themes with different characters or different rooms in a house. Niche-market producers have not taken over interactive titles yet, so be weird!

As you prepare sound effects, be mindful of the harmonic relations. If you use several effects simultaneously, they should harmonize, even if they are not music per se. For example, the jangling of keys, the clanging of a metal door, or a music cue have harmonic components you need to consider in relation to other sounds. The harmonization should not be limited to pleasing tones—at times, a tritone lends that perfect touch—but you should know all the possible intervals in your title.

Finally, when designing sound effects, decide carefully whether ambient sounds or musical cues should indicate events. For example, when users choose the hardest level a voice might say "Be careful!" A musical fragment or applause might indicate success. A bee flying might elicit "The Flight of the Bumble Bee" or a buzzing sound.

Tweak, Tweak, Tweak!

As your title goes to production, make sure music and sound effects are just right. Insist on early installation of sounds and music, and consider the following:

- ◆ **Volume settings.** How often is a sound effect heard and what should be its relative level? Normalize samples to end up with as little wasted headroom as possible. Compare overall title volume levels to normal TV listening levels.

- ◆ **Synchronization.** Use 3DO Animator and/or SMPTE to synchronize sound and pictures. Sometimes sound should not start at the head of a sequence of animation frames. Depending on the action the animation depicts, a pre-roll or a delay may be better. This requires close coordination with the development team's programmers.
- ◆ **Harmonic relations.** Know the implied intervals as sound and music interact in your title.

Project Management

Audio should be considered early in title development. Leaving sound for L.A.S.T. (Late Audio Sounds Terrible) affects overall quality. This problem is well known among computer title composers and may be a holdover from film scoring. In film work, it is best to score only to a final cut. In the interactive title business, audio has historically been in third place, behind game-play and graphics. Both content providers and project managers need to remain committed to great audio since the 3DO platform allows for a rich sound environment and titles lacking good audio lose an important element of a multimedia presentation.

Content providers need to listen to the requirements of everyone else involved in the title. How much RAM is available? Do team members want audio in a certain format (SR, mono/stereo, compressed, 16-bit or 8-bit)?

Project managers in turn need to set clear guidelines and veto rights on audio approval. Auditioning of new audio material needs to be handled consistently. Try to bring in new material in a similar way and ask that team members pay close attention as you audition new sound effects or music. Here are some tips:

- ◆ *Never reject **uninstalled** music and sound effects.* Magic happens when pictures fit with sounds.
- ◆ Keep in mind that the frame size of the picture to which audio is synchronized influences people's opinions. For example, if you are auditioning audio for an eventual Cinepak sequence, using Premier to synchronize audio to a QuickTime movie, people may think the audio is too powerful for the movie, because of the tiny QuickTime window on a Macintosh monitor.
- ◆ Consider using a paper form to track project progress. In-house developers at The 3DO Company use the "Audio Request, Status, and Critiques Form," shown on the next page, to facilitate this.

Audio Requests, Status, Critiques *(Updated Frequently)* **Date:**

Name of Title	Description of Sound Requested What it should sound like, or what it should be called.	Event or Graphic that the sound will accompany? Is it only off- screen – ie., no specific picture timing/sync?	Artwork (A) or Title (T)? Is this event now Artwork only, or is it in a working Title?	Status To be completed by Audio Guys: WIP, Ready (file/path), installed or Approved.	Critique/Comments on installed sounds and music. Enter comments in appropriate row. Completely optional. Thanks!

Installing and Launching ARIA

Introduction

ARIA is a Macintosh application for sound designers and others interested in exploring the 3DO™ system's powerful audio features. Its elegant interface encourages creative audio design. With ARIA you can set up a MIDI project window:

- ◆ To play back and debug MIDI files from disk on the 3DO Station.
- ◆ To get the 3DO Station to respond to real-time MIDI events from a sequencer or keyboard (for original music composition), using sampled or synthetic sounds.

An on-screen oscilloscope lets you view the sound on a video monitor, and an audio monitor screen accessible with the control pad shows DSP usage.

You can install the ARIA tool using the Easy Install or the Custom Install option of the installer program on the Toolkit CD-ROM.

System setup necessary after installation is explained in this chapter, which ends with a brief description of installing ARIA without using the installer program.

Chapter Overview

This chapter discusses the following topics:

Topic	Page
Hardware and Software Configuration	12
Launching and Setting Up ARIA	12
Setup for Working With MIDI	13

Topic	Page
Drag and Drop ARIA Installation	14

Hardware and Software Configuration

ARIA requires:

- ◆ Macintosh II or better
- ◆ System 7.1 or better, Sound Manager 3.0
- ◆ Comm3DO, installed somewhere on your hard drive

Launching and Setting Up ARIA

The following steps take you through starting the ARIA application and setting it up correctly so that you can use it.

1. Double-click on the ARIA icon to bring up the application. When you do this, ARIA automatically launches the Comm3DO application, if it is not already running.

Note: When you start ARIA for the first time, it creates a number of files and folders on your hard disk. If you expect to use samples to play MIDI, you need to move some files and folders into those newly created folders, as discussed later in this chapter.

2. When the Communications Preferences dialog appears, choose the correct bootscript, ROM, and video type, and click OK.
3. If ARIA launches successfully, the following windows appear:

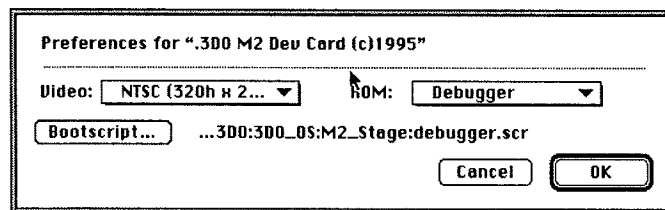


Figure 2-1 Communication Preferences dialog.

Note: The first time you open ARIA, the Communication Preferences Bootscript path name...3DO:etc will read "?"; the second time you use ARIA, your previous choice will appear automatically as you see in Figure 2-1.

Troubleshooting

If ARIA fails to launch successfully:

- ◆ Check the ARIA Log window (open it under the Windows menu) to see what went wrong and try to correct the problem.
- ◆ Check the Comm3DO application console window to see if problems were reported there.
- ◆ Throw away the *ARIA Preferences* folder (*System/Preferences/3DO/ARIA Preferences*).

Setup for Working With MIDI

This section explains some system setup for working with MIDI. It is recommended you go through the setup now since ARIA requires MIDI support.

Files and Folder Setup

If you expect to use samples to play MIDI, you need to copy some files into folders that ARIA creates the first time it launches successfully.

To move the files and folders, go through these steps:

1. If you haven't already done so, launch ARIA, check to make sure that all is well, and then quit ARIA.
2. In the *3DO/remote* folder, locate the following folders created by ARIA:
 - ◆ */remote/ARIA/Instruments*
 - ◆ */remote/ARIA/Samples*
 - ◆ */remote/ARIA/MIDI*

If you want to run the example MIDI program, add the following two steps.

3. Drag the contents of the *ARIA MIDI Examples* folder (the *MyGroove* folder) into the new *MIDI* folder.
4. Drag any samples that you expect to use from the Toolkit CD-ROM to the *Samples* folder.

Required Files for Real-time MIDI

If you intend to play back standard MIDI files from disk, the folder structure described above is sufficient.

To create music using ARIA's real-time MIDI capabilities, you also need the following:

- ◆ Apple MIDI Manager, including PatchBay, and the Apple MIDI driver, if it is needed by the sequencer. Apple MIDI Manager is available from the Apple Developer's Association.

- ◆ A MIDI sequencer compatible with the MIDI Manager.
- ◆ A MIDI keyboard.

Setting Up for MIDI

For real-time MIDI playback from a sequencer, or to play a patch from a MIDI keyboard, you will need the following:

- ◆ Apple MIDI Driver in your *System* folder.
- ◆ Apple MIDI Manager in your *Extensions* folder.
- ◆ PatchBay application. Consider placing it in your *Apple Menu Items* folder for easy access.

These programs are available separately and are usually bundled with sequencer applications.

Figure 2-2 shows the PatchBay window set up for all possible uses with ARIA.

The individual connections have the following effect:

- ◆ Serial port to sequencer—used for composing with a sequencer.
- ◆ Sequencer to ARIA—real-time playback from sequencer.
- ◆ Serial port to ARIA—playback of patches from MIDI keyboard.

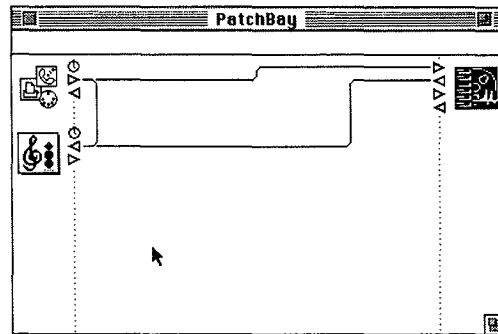


Figure 2-2 PatchBay setup for working with ARIA.

Warning: If you are using all three of these connections, turn off the keyboard "thru" option in the sequencer to avoid "note bounce."

Drag and Drop ARIA Installation

To install ARIA without the installer:

1. Find the ARIA folder on the CD-ROM; it contains the ARIA application and an ARIA Examples folder. In addition there are patch documents and some MIDI examples.
2. Copy the ARIA application to your hard disk.
If you want, you can also copy the examples.
3. Go through the steps in the section "Files and Folder Setup" on page 13.

Note: *AIFF samples are not System 7 snd_files and therefore cannot be previewed with Sound Manager 3.0. Use SoundHack to listen to them instead.*

Creating MIDI Projects With ARIA

Introduction

This chapter explains how to use a 3DO PIMap (program-to-instrument map) file in ARIA to play standard MIDI files already saved on disk and how to generate a general MIDI PIMap (a new feature of ARIA 2.0D) to play MIDI files. You also learn how to use ARIA in conjunction with a MIDI Manager-compatible sequencer to create music in real time.

Chapter Overview

This chapter discusses the following topics:

Topic	Page
Preparing Your Environment	18
MIDI Files and PIMap Files	18
Playing a MIDI File on the 3DO Station	20
Working With General MIDI	22
Working With Real-time MIDI	23
Tips and Tricks	23

Preparing Your Environment

Before you can start to work with MIDI files, you need to set up your system correctly:

1. Make sure your environment is set up as described in the section “Drag and Drop ARIA Installation” on page 14.
2. Check that the appropriate files and folder exist, as described in the section “Files and Folder Setup” on page 13.
3. Launch Comm3DO and then launch ARIA, as described in the section “Hardware and Software Configuration” on page 12.

Available Example Files

The following files are available on the Toolkit CD-ROM:

- ◆ A number of *.aiff* samples in the *AIFF Samples* folder.
- ◆ A PIMap file, a sample, and a MIDI project file in the ARIA Examples folder.

MIDI Files and PIMap Files

To play a MIDI file on the 3DO Station, you need the following files:

- ◆ The MIDI file itself
- ◆ The PIMap (program-to-instrument map) file, which tells the 3DO system how to match MIDI numbers with sounds
- ◆ Compiled patches (*.ins*) or samples referred to in the PIMap

This section discusses the PIMap file format, including the currently supported flags and an example file.

Note: For more information on PIMaps, see the *3DO M2 Music and Audio Programmer's Guide*, Chapter 9, “Playing MIDI Scores.”

Working With PIMaps

A PIMap associates MIDI program numbers with 3DO instruments.

Parts of a PIMap

A PIMap consists of several lines, separated by carriage returns. Each line contains:

- ◆ MIDI program number (1–128)
- ◆ Filename

- ◆ If the filename ends in .aiff, .aif, or .aifc, it is considered to be the name of a sample file and the file is attached to the appropriate sample player instrument.
- ◆ If the filename does not end in .aiff, .aif, or .aifc, it is assumed to be an instrument name. For example, if the file ends in .ins, it is a compiled ARIA patch.
- ◆ Optional flags

Currently Supported Flags

Table 3-1 lists the flags that are currently supported as part of a PIMap:

Table 3-1 *Flags in a PIMap.*

Flag	Description
-f	Play sample file using a fixed-rate sample player (such as a drum) that uses fewer ticks (DSP resources) and less DSP memory than a variable-rate player would.
-m <i>n</i>	Set maximum number of voices (default is 1).
-l <i>n</i>	Set low note for a sample.
-b <i>n</i>	Set base note for a sample.
-h <i>n</i>	Set high note for a sample.
-p <i>n</i>	Set priority for instrument, default = 100, max = 200.
-d	Set a detune value. This works only on instruments that use the default sample playback module (sampler.dsp), not on compiled ARIA instruments.

Note: *The base note is the “home” note of the sample and is not necessarily the lowest (bass) note.*

PIMap Example

Example 3-1 shows a small PIMap file. The following points are of interest:

- ◆ The PIMap file assigns the appropriate samples to the different ranges of the instrument.
- ◆ Each .aiff file covers no more than a half octave, as you can see when you look at the low notes, bass notes, and high notes assigned to the different .aiff files.
- ◆ The gong is fixed pitch.
- ◆ The clarinet is set up as a multisample with octave spacing.
- ◆ The organ has its MaxVoices set to 4 to make it possible to play chords.

- ◆ The sample *bell.aiff* is played by the instrument *Sampler_16_v1.dsp* because the instrument is specified before the sample is specified.
- ◆ The instrument ending in *.ins* (*windnoise.ins*) is a compiled ARIA instrument.

Example 3-1 *PIMap example file.*

```
1 clarinet1.aiff -l 30 -b 48 -h 53
1 clarinet2.aiff -l 54 -b 60 -h 66
1 clarinet3.aiff -l 67 -b 72 -h 84
2 gong.aiff -f
3 bass.aiff -p 150
4 organ.aiff -m 4
5 sampler_16_v1.dsp
5 bell.aiff
7 windnoise.ins
```

Playing a MIDI File on the 3DO Station

To play a MIDI file on the 3DO Station using ARIA involves importing a PIMap, then loading and playing the files, as explained below.

Importing the PIMap File Into a MIDI Project Window

To paste the PIMap file into a MIDI project document, follow these steps:

1. From the ARIA File menu, select New MIDI Project or use Command-M.
2. Click the MIDI file button and load a MIDI file from the file selection dialog that appears.
3. From the MIDI menu choose Import PIMap. The result is shown in Figure 3-1. If you have already located the MIDI file, you could also select Import and Optimize PIMap, which checks the MIDI file and discards unused lines in the PIMap. This is discussed in the next section.

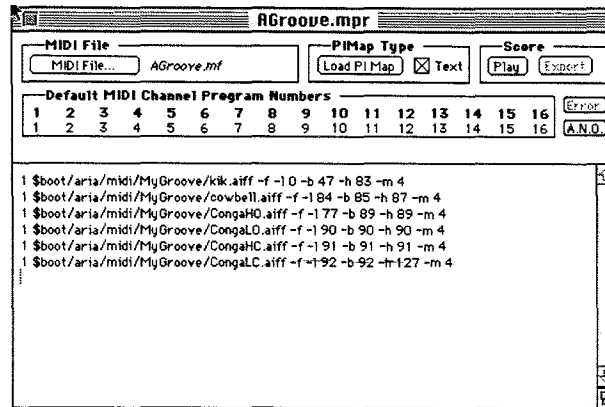


Figure 3-1 MIDI project with PIMap file.

Loading and Playing the Files

Once your PIMap file is in the MIDI project window, you have to tell ARIA where to find the MIDI file, then load the PIMap file.

To load the files, follow these steps:

1. To locate the MIDI file, click on the MIDI File button in the top-left corner of the MIDI project document and select the MIDI file you want to play.
2. Finally, click the Load PIMap button, which is also in the top row to the right of the MIDI File box.

Note: Loading the PIMap and MIDI file takes a while. Progress is reported in the Comm3DO window. If you bring the Debugger Terminal window to the foreground while loading the PIMap, the procedure runs more quickly.

3. When the PIMap finishes loading, click on the Play button in the MIDI project box to load the MIDI file.
4. Click the Play button (single arrow) in the dialog that appears to play the MIDI file on the 3DO Station.

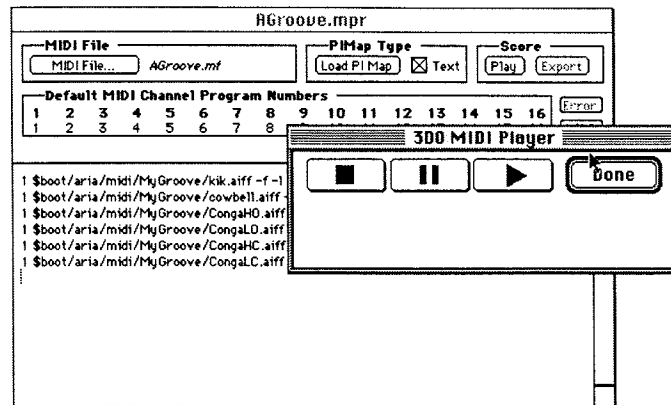


Figure 3-2 MIDI project window with 3DO MIDI Player dialog box.

5. If you have problems, check the volume on your TV monitor and adjust it if necessary.

Caveats

- ◆ MIDI files will not necessarily play on the 3DO system without some tweaking. The 3DO system has limited DSP resources, so without optimization, using fixed instruments, and so on, the average number of simultaneous voices available is six.
- ◆ The DSP limits you to playing a theoretical maximum of 32 samples simultaneously; ARIA limits you to 12.
- ◆ 3DO samples vary in memory size and ticks. See the Audio docs for DSP instrument resource requirements. We encourage you to use SQS2 for your mono samples. The results sound good and no extra resources are required to place the compressed samples. The samples take only one half of the memory used by 16-bit samples.

Working With General MIDI

If you're working with general MIDI, which is a standard built on top of MIDI, with standard programs, you don't need to manipulate the PIMap. The reason is that general MIDI specifies what all 128 MIDI program numbers (0–127) mean; that is, it has already mapped the program numbers to sounds.

The advantage of working with general MIDI is that it saves both mapping time and PIMap load time. It also saves memory by giving you only as much of a PIMap as you need.

To generate a general MIDI PIMap:

1. Make sure you've installed MIDI samples inside your *ARIA/MIDI* folder.
2. Choose Open New MIDI Project Window under the File menu, or use Command-M.
3. Click on the MIDI file button in the upper-left corner of the window and use the file selection dialog that appears to open your MIDI file.
4. Select Generate General MIDI PIMap from the MIDI menu.
5. ARIA reads the notes in the MIDI file and generates a PIMap sufficient for all notes in the file.

Working With Real-time MIDI

While ARIA is useful for playing MIDI files on a 3DO Station, you will find it even more helpful for editing MIDI files in real time. Since the actual process of editing the MIDI files depends on the sequencer you have available, this section only gives a brief overview of the process:

1. Connect the Output of the MIDI sequencer to the Input of ARIA, using the Apple MIDI Manager's PatchBay application. Be sure that you have the Apple MIDI driver in your System folder and the MIDI Manager extension in your Extensions folder. The PatchBay application should be in your Control Panel.
2. Click the PatchBay icon on the left for MIDI driver settings.
3. Go back to ARIA and choose Open New MIDI Project from the File menu.
4. Load the MIDI file you want to edit into the MIDI sequencer (or start composing a new document).
5. Connections appear in the PatchBay window.
6. Additional windows in the sequencer application allow you to edit MIDI data.

Tips and Tricks

This section first provides some additional hints on working with the MIDI project interface, then some more general advice.

MIDI Project Interface Tips

- ◆ **Default MIDI Channel Program Numbers.** If you don't have program changes in the MIDI stream, ARIA uses the default. You can assign a channel (top row) to a program (bottom row) by replacing the program number with a channel.
- ◆ **ANO.** Turns off hanging notes after you stop a MIDI file.

- ◆ **Error.** This button is dimmed if no errors occur, but flashes if there is an error. You can then click it to view the error information in a dialog box.

MIDI Tricks

- ◆ Playing a patch via real-time MIDI produces drastically different volume levels depending on whether the patch window or the MIDI project window is selected.
- ◆ Program numbers in a PIMap range from 1–128. When the program number for an instrument in a PIMap is interpreted, it is decremented by 1 to make the range 0–127 instead of 1–128. When using a sequencer to prepare a MIDI file to use with the PIMap, be sure to set its program number range accordingly, or offset the program numbers in the PIMap by one.
- ◆ When setting up one or more instruments on a particular MIDI channel in the PIMap, make sure that the note ranges are set up so that all notes on that channel can trigger something, even if only a small note range is to be utilized. If a note-on instruction is broadcast on a channel that does not actually trigger a note for the instrument, no further notes will be heard until the debugger and ARIA are relaunched. MIDI information is still being received in this state; it is possible to have it displayed in the Debugger Terminal window, but it does not generate any events that can be heard. MIDI files already loaded in the MIDI project window, however, will still play after this state is reached.
- ◆ ARIA allows 12-note polyphony for real-time playback. Levels are set to three times the safe level (one-twelfth of maximum on each channel). This causes clipping if more than four voices play at maximum amplitude at exactly the same time.

SquashSnd

Introduction

SquashSnd is a self-documenting MPW tool for compressing and decompressing audio samples. The 2.1 and later versions of SquashSnd are 5-to-25 times faster than previous versions. For basic usage information, open MPW on your Macintosh, type SquashSnd (with no parameters), and press Enter.

This chapter discusses the following topics:

Topic	Page
SquashSnd-Compatible File Types	25
SquashSnd Parameters	26
Caveats	27

SquashSnd-Compatible File Types

SquashSnd accepts files in Audio-IFF (AIFF uncompressed) format or Audio-IFC (AIFC uncompressed) format. The output file for AIFF is AIFC, but AIFC output depends on the content of the input file. For AIFC files that contain uncompressed data, the output file is AIFC. AIFC files that contain compressed data should output to AIFF.

Input and output files may also contain raw sample data. To convert raw files, more parameters (discussed below) must be supplied to SquashSnd.

SquashSnd Parameters

SquashSnd parameters for compressing and decompressing sample and raw data files follow. Bracketed parameters are optional.

```
SquashSnd -i infile -o outfile  
-sqs2 | -cbd2 | -adp4 | -opera | -halfrate  
[-p] [-y | -n] [-play] [-ch chcount] [-expand] [-raw]
```

-i infile - name of input file

-o outfile - name of output file

-halfrate - 2:1 sample rate conversion (44K → 22K)

-expand - decompress a compressed file

If this parameter is omitted, compression is performed.

-ch chcount - number of channels (1-6)

This parameter is required for raw files. For an Audio-IFC or Audio-IFF file, this parameter may be included, but it must agree with the information in the file.

At most one of the compression types below may be used in a SquashSnd command. You must include a compression type for raw files and when compressing Audio-IFF/IFC. You may include a compression type when expanding an Audio-IFC file, but only if it agrees with the information in the file.

-sqs2 - 2:1 Squareroot-Delta-Exact shifted compression for mono only. Compatible with 3DO M2 hardware.

-cbd2 - 2:1 Cuberoot-Delta-Exact compression. Compatible with 3DO M2 hardware.

-adp4 - 4:1 ADPCM compression (DVI algorithm). Adaptive Differential Pulse Code Modulation. Compatible with 3DO M2 and earlier development system hardware.

-opera - 2:1 Squareroot-Delta-Exact compression, compatible with 3DO development system hardware prior to M2.

The following parameters are optional for all file types:

-p - output progress information

-y - answers "yes" to dialog about replacing existing file. Avoids dialog.

-n - answers "no" to dialog about replacing existing file. Avoids dialog.

-raw - force output file to be raw sample data, regardless of the input file type

-play - play expanded Audio-IFF/IFC file if Sound Manager 3.0 or later is present.

Note: *SquashSnd 2.1 is a "fat" MPW tool. This means that as long as your MPW Shell and/or tool server are PPC savvy (MPW Shell v.3.4 or later), launching SquashSnd will automatically run native code on PPC and 68K code on a 68K machine.*

Caveats

- ◆ Problems may occur if -ch chcount is set to an incorrect value. For example, if the raw file has two tracks and -ch chcount is set to 3, the output will be garbage. You will get no warning because SquashSnd has no way to determine if the given information is accurate.
- ◆ If you compress raw files with one type of compression you must decompress with the same type. If you do not, the operation will not work, and you will get no warning or diagnostic message.
- ◆ An obscure problem occurs when using SquashSnd with ToolServer. If you are running an MPW script file that repeatedly tells ToolServer to run SquashSnd, thus

```
rshell -b "SquashSnd . . ."  
rshell -b "SquashSnd . . ."
```

and if you use Command - . to stop processing, your Macintosh will crash. To avoid this problem use the MPW shell rather than ToolServer for this type of batch compression.

- ◆ You cannot do both -halfrate and another compression in one squashing. This feature may be allowed later. To do two, or more, compressions now you must run SquashSnd two, or more times.

3SF Overview

Introduction

This chapter presents an overview of the 3DO Score File Format, and the architecture for its supporting software.

The chapter covers the following topics:

Topic	Page
The 3DO Score File Format	30
3SF Architecture	30
Before 3SF	30
Sound Design With 3SF	32
Score File Creation	32
Score File Playbackscore	33
	33

The 3DO Score File Format

The 3DO Score File Format (3SF) is an all-in-one format for real-time playback of sound effects and music from a 3DO title. A 3SF file can contain multiple data types, including embedded MIDI files, MIDI file fragments, sampled sound data, synthetic instruments for the 3DO digital signal processor (DSP instruments), and configuration parameters.

A 3SF file can represent a fixed horizontal or linear composition (one derived from a single sampled-sound file or a single MIDI file) which is played from beginning to end without interruption. Taking this one step further, a 3SF file can add to such a composition the capability of looping or branching within the file.

By contrast, a 3SF file can also represent a vertical bank of sounds and music sequences, selectable simultaneously or separately under program control. Individual sounds can be selected by name and played without regard to their type.

3SF Architecture

The 3SF Architecture is the architecture of the supporting software, which you can use to create, process, and play back a 3DO Score Format File.

The 3SF Architecture addresses some of the resource trade-offs incurred with earlier methods of storing and playing sounds. If memory is constrained, a 3SF file allows demand-loading and spooling from disk. If disk access is restricted or forbidden, a 3SF file can be read in initially and become memory-resident.

This chapter describes the intended goals of the 3SF Architecture from a high level. Its purpose is to make clear the general class of features provided, what kinds of data are used as inputs and outputs, and the relative priorities of the various features. Implementation is covered later in Chapter 7, "3SF User's Guide."

This document assumes familiarity with sound samples, MIDI, the 3DO Audio folio, and the 3DO Music Library.

Before 3SF

Prior to inclusion of the 3SF Architecture, 3DO title developers had several options for playing sounds and music on the 3DO System, each with relative advantages and disadvantages. In some cases the programming steps needed to implement these options were quite complex. Worst of all, the code needed was different among the various methods.

Method 1: Play a Sample in Memory

This is the simplest method, which involves reading a sample file (in AIFF or AIFC format, or .aiff files) into 3DO memory and playing it back using an appropriate program call. This is a memory-intensive operation, because high-quality samples can be large, even if compressed, but it does avoid disk access.

Method 2: Spool a Sample From Disk

This is similar to playing a memory-resident sample file, except that 3DO memory is saved at the expense of disk access. Spooling (i.e., sequentially reading small amounts of data from the file, just in time for playback) is best accomplished by making calls to the Advanced Sound Player interface, which actually supports spooling, looping, and branching, even between files. For a RAM-resident title, with little memory allotted for audio, spooling was formerly the only practical method to play music.

Method 3: Weave a Sample Into a Stream

This method, somewhat analogous to Method 2, integrates the sound track with the other data. All the data, including audio, is “woven” using the weaver, and “streamed” in at run time. Streamed data is read using the Audio Subscriber.

This technique takes advantage of the strengths of data streaming: only the audio being played needs memory, and only a single disk file is needed, which limits seeking. The disadvantages include the complexity of data streaming, and the fact that the music will be synchronized to the rest of the stream content.

Method 4: Play a Synthetic Patch in the DSP

This method is ideal for simple sound effects such as sirens, helicopters, laser blasts, wind sounds, and telephones. It involves loading a DSP Instrument (a small program) into the DSP, and playing it via a program call. A number of compiled DSP instruments, stored in .dsp files are provided as part of the 3DO Portfolio. You can make your own DSP instrument using MakePatch.

Method 5: Play a MIDI File

This method requires first loading a PIMap (program-instrument map) into memory. This loads all the required sample data into 3DO memory and all the required synthetic instruments into the DSP, typically consuming a great deal of memory. The industry-standard MIDI file itself, which is fairly compact, is spooled in from disk and played, event by event, by program calls. Music of long duration can be stored much more compactly via MIDI than as sampled data, because one sample can represent many notes. The main drawback is that the associated data has to be memory-resident and must be preloaded.

MIDI sequences can use samples (.aiff files) and both kinds of synthetic instruments (.dsp and .ins). Moreover, a MIDI PIMap, once loaded, can be used as a sound-effects engine, since a single sound can be associated with each pitch, and played with a simple call.

Sound Design With 3SF

A single 3DO Score File (3SF file) can contain all types of sound data playable on the 3DO station, including sequencing information to indicate how the file is to be played. Files are playable, start to finish, with a single call.

The 3SF architecture offers the following benefits:

- ◆ For horizontally playable files, one call plays an entire file, whether it contains samples, MIDI data, synthetic instrument definitions, or a complicated sequence that combines the above.
- ◆ The 3SF file format can optionally be an all-in-one format (though external references are allowed). MIDI files, PIMaps, sample files, and synthetic instrument files are combined into one, (and in some cases the original files, can usually be retrieved from the 3SF file), with the following benefits:
 - ◆ The file is transportable. Since it is self contained, it is easy to move from one system to another; there are no aliases to patch up. One file contains everything needed for playback.
 - ◆ It simplifies programming for playback, as a few simple calls will load, prepare, and play all the disparate ingredients needed for a title, based on instructions within the file. The gain over existing MIDI playback is considerable.

Score File Creation

There are currently two methods available to 3DO title developers for 3SF file creation:

From ARIA

A 3SF file based on a MIDI sequence can be created in ARIA from the MIDI project window. This window currently assembles all the ingredients needed to make such a 3SF file. An ARIA menu command "Export 3DO Score File..." lets you make a 3SF file. It gathers up the MIDI sequence, the PIMap, and all of the files referenced by the PIMap, and stuffs them into a single Score File. (Note that certain system instruments that are always present need not be copied into the Score File, but only an appropriate reference to them.)

MakeScore Tool

You can also use the MakeScore tool (an MPW tool) for score creation. It provides a means of automating the Score File production process, since ARIA is not scriptable. The MakeScore tool is also capable of creating 3SF files more general than those that ARIA can create.

The MakeScore tool is invoked from the command line using the following syntax:

```
MakeScore [-p] [-x] -map <PIMap file> -midi <MIDI file>
```

The `-p` option specifies that progress information is to be printed.

Using the `-midi` and `-map` options, MakeScore creates a simple 3SF file that combines the MIDI and PIMap data into a single file. Samples referred to in the PIMap are included in the file unless the `-x` option is used.

Tools for other platforms may become available in the future.

Score File Playbackscore

There are three ways to play a Score File, all of which allow you to hear the sounds played on the 3DO hardware:

From ARIA

ARIA plays a 3SF file via the "Play Score File..." menu command. This command plays any 3SF file designed for horizontal playback, even those not created by ARIA. Sound designers will prefer this method.

PlayScore

PlayScore is a 3DO application that lets you load and play a 3SF file, using the controller to select, interact, and branch, where appropriate. PlayScore allows you to do pause and resume playback, or rewind to the beginning using the control pad.

From a C or C++ Program

An interface (`sfplayscore.h`) and a linkable library (`sfplayscore.lib`) are available for playing a 3SF file from a C program. This interface allows preloading, playing a file, or a resource from a file by name.

3SF Architecture

Overview

This chapter gives a high-level view of the 3SF architecture for creating and playing 3SF score files. It explains the developer-level code organization and interdependencies.

The PlayScore interface, is provided for playing files on the 3DO. Both APIs are supported by linkable libraries. The interface files and libraries of PlayScore are shipped to licensed 3DO developers on the M2 CD.

The API is usable either from C or from C++. Conforming with convention, the compile-time flag `_cplusplus` is used to shield a mere C compiler from encountering C++ code.

Because of this architecture, it should not be necessary for developers to know the format of a 3SF file.

This chapter covers the following topics:

Topic	Page
Creating a 3DO Score File	36
Playing a 3DO Score File	37

Creating a 3D0 Score File

As described in Chapter 5, “3SF Overview,” two high-level tools for sound designers, ARIA and MakeScore, can create 3SF files under developer control. In point of fact, this tool conforms to the rule of operating through the `sfmakescore` API.

The following diagram illustrates this relationship:

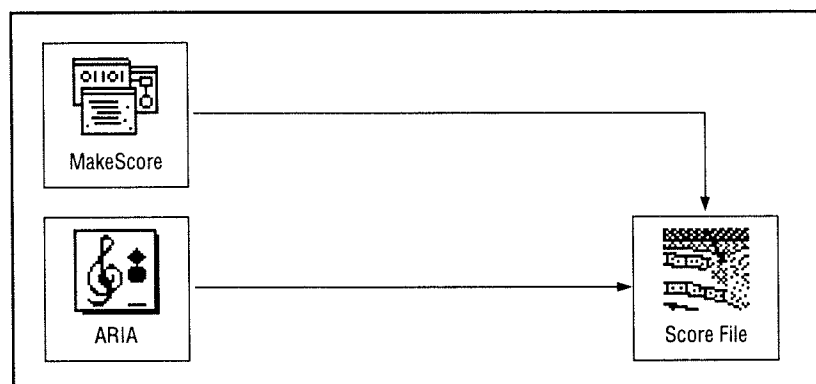


Figure 6-1 Creation of a 3SF file.

The `sfmakescore.h` interface has the following calls for C++ programmers:

```
sfMakeFile( const PSSpec& standardMIDIFile, // MIDI file
            const PSSpec& outputFile, // output file
            const PSSpec& pimapFile); // PIMap file
```

This creates a 3SF file from a MIDI file and a PIMap file.

To create a 3SF file from a MIDI file and a string defining the PIMap, use the following calls:

```
sfMakeFile( const PSSpec& standardMIDIFile, // MIDI file
            const PSSpec& outputFile, // output file
            const char* pimapString); // PIMap text
```

The `MakeScore.h` interface has corresponding calls `sfMakeFilePMSpec` and `sfMakeFilePMString` for C programmers using pointers instead of references.

Playing a 3DO Score File

Playing 3SF files is ultimately done through calls to functions in the *sfplayscore.h* interface file. The implementation is written in portable style, though the initial implementation has been done for the 3DO system.

For sound designers, this code is packaged in the high-level ARIA and PlayScore tools.

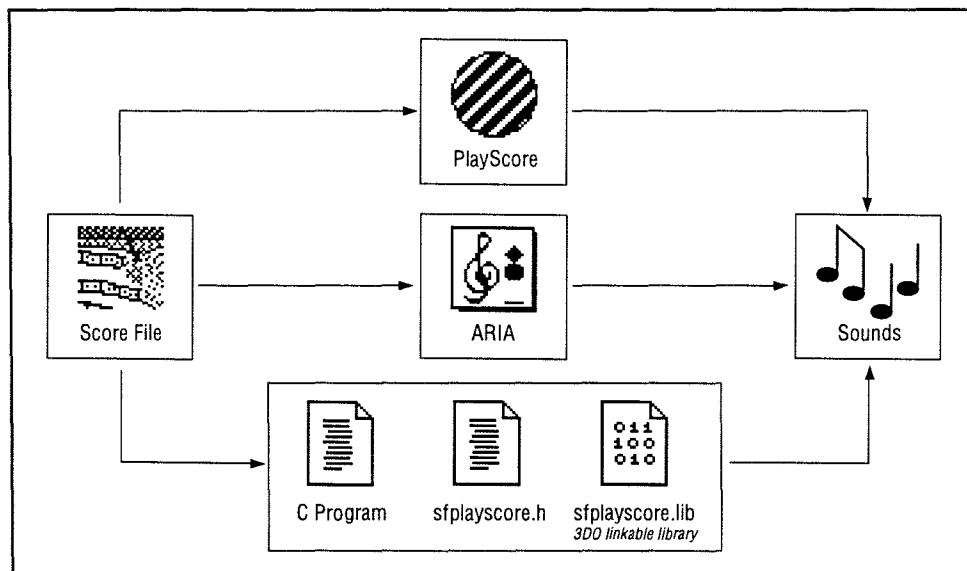


Figure 6-2 *Three ways to play back a 3SF file.*

3SF User's Guide

Introduction

This chapter discusses how to use the 3SF audio file format in sound design. It covers the following topics:

Topic	Page
Why Use 3SF	40
The 3SF Tool Set	40
PlayScore.lib	43

Why Use 3SF

As previously mentioned, the 3DO Score File Format, or 3SF, combines a variety of audio files into a single file. The benefits of doing this are

- ◆ Speed-up of the development procedure, by allowing the sound designer to deliver a single, self-contained file to the programmer.
- ◆ Efficient loading at runtime, because much of the loading analysis is carried out when the file is built, and because disk seeks are optimized by keeping the loaded data contiguous.
- ◆ Ease of programming, because the programmer can play a 3SF file from "C" with just a couple of simple C calls.

This release provides tools for creating a 3SF file that combines all auxiliary files (MIDI file, PIMap, and sample files) needed to play a single MIDI score.

Sound designers using 3SF will typically develop a MIDI score from a favorite sequencer program, develop a PIMap with ARIA, and then package the score and sample files into a 3SF file. This single file can be delivered to the programmer, who will be able to play it back with a few simple calls from a C program.

Note: *It's possible to have a 3SF file (containing a MIDI score) playing in the foreground, with looping audio in the background.*

The 3SF Tool Set

The 3SF tool set consists of three tools:

- ◆ MakeScore creates files to play on the 3DO system.
- ◆ DumpIFF dumps the contents of a 3SF file.
- ◆ PlayScore plays the file on the 3DO system.

Both MakeScore and DumpIFF are MPW tools; PlayScore is a 3DO application.

MakeScore

Install MakeScore in your *3DO:Tools* folder. You must also have MPW installed on your Macintosh. For usage instructions type MakeScore <enter> with no command line options in the MPW window. The following instructions appear:

```
# makescore:
# Creates a 3DO Score Format File (3SF file)
Usage - MakeScore [-o outfile] [-p] (-midi midifile -map mapfile)
# Options:
    outfile  Name of output file.
    midifile  Name of standard MIDI file.
    mapfile   Name of PIMap file.
    -p       Output progress information.
```

MakeScore takes an optional output file determined by the `-o` option. If the `-o` option is not specified, MakeScore names the file "score.3sf." Note that MakeScore, like all MPW tools, overwrites any existing output file without warning.

The `-p` option gives progress information during the score making process. The user can specify two `-p` options for ultra verbose mode.

The input files must be both a MIDI file and a PIMap file. MakeScore combines the MIDI file and the samples referenced by the PIMap file into the output file. Typically MIDI files end in `.mf`, and PIMap files have the `.pmp` or `.pimap` suffix. For examples, look in *Examples:Audio:Songs*.

An example MakeScore command line looks like this:

```
MakeScore -midi '3DOPortfolio  2
2.5:Examples:Audio:Songs:Zap:zap.mf' 2
-map '3DOPortfolio  2
2.5:Examples:Audio:Songs:Zap:zap.pimap' 2
```

MakeScore processes the PIMap by loading each sample referenced and including it in the output file. Before you can run MakeScore, certain files and folders need to exist. These will be installed automatically if you correctly perform the M2 CD installation. The most important requirement is that your remote folder contain a subfolder called "ARIA," which in turn contains your "Samples" folder. For more information, see "Setup for Working With MIDI" on page 13.

The MPW shell variable `{3DORemote}` should refer to the same */remote* directory that contains the directory that ARIA has saved its files to. You can switch remote folders by using the 3DO menu in MPW. If you don't have this menu available you may not have installed the 3DO M2 2.0 SW CD correctly.

MakeScore interprets the alias *\$samples* to be *{3DORemote}ARIA:Samples*, so you can use *\$samples* in the pathname of your samples. An example of this would be *\$samples/PitchedL/Viola/Viola.C3LM44k.aiff*.

If you do not have the referenced samples installed you may get an error message like the following:

```
Could not find path
:aria:samples:pitchedl:electricpiano:electricpiano.c3lm22k.aiff in
remote folder
Recovering from failure...
# Directory not found (OS error -120) (-120)
# Since errors occurred, score.3sf will not be written.
```

If all the required samples are present, MakeScore includes them, and the MIDI sequence, into a self-contained output file. To play the file, copy it to your remote folder so it can be accessed by the 3DO system.

Note: *Your Macintosh System folder/Extensions/3do contains a 3DO folder and a 3DO Release folder. These files can be edited to point to the correct directory.*

DumpIFF

DumpIFF produces a symbolic dump of a 3SF file, or any IFF file. This operation reveals such information as the content of chunks, sample rates, and so on; you might run DumpIFF in order check all the contents of a file you're having problems with. DumpIFF parses the IFF file, checks to see that the files are valid for their format, and produces a listing of the contents in the MPW worksheet. An example of the output from DumpIFF appears in Appendix B, "DumpIFF Sample Output."

Like MakeScore, DumpIFF is an MPW tool that should be copied into the *3DO:Tools* folder. Usage instructions are available by typing DumpIFF from the command line in MPW:

```
# DumpIFF <filename> :
# Dumps the contents of an IFF file or ThreeSF file.
Usage - DumpIFF infile [-p]
# Options:
      -p                Output progress information.
```

DumpIFF takes a single required parameter: the name of the input file. A -p parameter to monitor progress is optional. Output can be redirected to a file using MPW's redirection symbol (>), which lets you load the output file and view it at leisure. The following is an example of a DumpIFF command:

```
DumpIFF score.3sf > myOutput
```

If the source file is not an IFF file an error message is returned.

PlayScore

PlayScore is a 3DO application, which must be run from the Debugger or the Comm3DO application. Copy PlayScore and a Score file to play into the remote folder. Then launch 3DODebug. From the command line in the Debugger Terminal window, or the Comm3DO console window, type:

```
PlayScore CoalRiver.3SF
```

and press the Enter key. The following message appears:

```
Awaiting control-pad commands. Press C for help...
```

Pressing C yields the following instructions:

```
Start/Pause (P) = Load and play the file (if not playing).  Pause/
Resume          (if playing)
Stop (X) = Stop playing.  Press again to exit program
C = Control-pad help
```

Press the play button to play the score file. If you spelled the file name incorrectly a message like the following one appears:

```
# Couldn't open input file DoesntExist
# Since errors occurred, DoesntExist will not be played.
```

Otherwise the file will be loaded and then played. During playback you can press the Play/Pause button to pause playback, or press the Stop button to stop playback. Once stopped, pressing play causes the file to start playing from the beginning, and pressing stop exits the file.

PlayScore.lib

You can play score files from your own 3DO application by linking with the PlayScore library `libplayscore.a`. The `PlayScore.h` header file describes the routines available in `libplayscore.a`, and how to call them. A brief description is given here.

ScorePlayer

In order to initiate the player, you must first create a an `SFScorePlayer`. To do this, call:

```
SFScorePlayer* sfCreateScorePlayer(short verbosity, SFErrProc
errProc);
```

The first parameter is verbosity level, which should be in the range 0–2. The second parameter is a pointer to an error procedure that you create. This pointer can be `NULL` if you don't want an error procedure called, or you can use the optional error procedure that appears below. `sfCreateScorePlayer` returns a

pointer to a 3SF private `SFScorePlayer` structure, and you must pass this pointer as the first parameter to the `sfStartReadingAndPlaying` routine in order for the file to be read, loaded, and played.

ErrorProc

The error procedure is optional. However, if errors occur during playback, the PlayScore Library needs to be able to call your host application. The error procedure you write should look like the one below, where `psErr` is an error code and `errText` is a pointer to the text string that returns:

```
void MyErrorProc(psErr err, const char* errText);
```

`err` is a 3DO error value (the type `psErr` is equivalent to the 3DO `Err` type, as defined in `pserrors.h`). Regardless of the value of the `err` parameter, your error function `MyErrorProc` must print the error text (if it is not null). Then, if the error value indicates a fatal error, you should handle it accordingly.

StartReadingAndPlaying

Playing the score file simply becomes a matter of calling `sfStartReadingAndPlaying` with the `SFScorePlayer` pointer, and a file path to open. The PlayScore Library then opens the file, reads in the data, and plays it.

```
SFFile* sfStartReadingAndPlaying SFScorePlayer* player const char*  
infilename);
```

When you finish playing files, just call `SFDeleteScorePlayer`, passing a pointer to your `SFScorePlayer` variable. The reader is disposed of, and your variable is set to `NULL`.

```
void sfDeleteScorePlayer(SFScorePlayer** Player);
```

StartReading

For finer control over the playback process, you need to load the entire file into memory by calling `sfStartReading` instead of `sfStartReadingAndPlaying`. This reads the data to be played, but defers playing any sound until a subsequent command (`sfStartPlaying`) is issued. Like `sfStartReadingAndPlaying`, `sfStartReading` returns an `SFFile` reference that identifies which file has been loaded.

```
SFFile* SsfStartReading(SFScorePlayer* player, const char*  
infilename);
```

Play, Pause, Stop

You may call `sfStartPlaying` to start playing the file, or `sfPause` to pause playback. `sfResume` resumes playback. `sfStop` terminates playback until another `sfStartPlaying` is issued. Executing `sfStartPlaying` while paused or playing will rewind to the beginning of the song and then start playing. When you are finished with the score reader call `sfDeleteScorePlayer`.

```
void sfStartPlaying(SFFile* file);  
void sfPause(SFFile* file);  
void sfStop(SFFile* file);
```


Essential Features of SoundHack

Introduction

SoundHack manipulates sound files, allowing you to convert monaural files to stereo, change pitch and sample rate, compress files, combine two files to create a new file, and so on. Its relative ease of use encourages experimentation.

Note: *SoundHack does not support the current 3DO sound compression formats. You need to use SquashSound for these functions.*

This chapter looks at the features of SoundHack that you are most likely to find useful: sample rate conversion, how to read and write raw files, and how to compress a file by saving it as AIFC. The remaining functionality of SoundHack is discussed in Chapter 9, "SoundHack User's Guide."

Programming skills are not necessary for using SoundHack, but you should be familiar with the Macintosh and with sound design.

This documentation pertains to SoundHack3DO 2.13, which runs on Macintosh II machines and above with floating point. The M2 Toolkit CD also includes non-floating point and PPC-native versions of SoundHack.

Chapter Overview

This chapter discusses the following topics:

Topic	Page
Installing SoundHack	48
Converting a Sound File to Compressed Format	48
Reading/Writing Raw Files	49

Topic	Page
Changing Header Information	50

Installing SoundHack

SoundHack is included on the M2 2.0 SW CD-ROM. It appears in the Audio Tools folder in your 3DO folder. If you have not already installed the M2 SW CD on your system, see *Getting Started With 3DO 64-Bit Development Environment Release 2.0*.

Converting a Sound File to Compressed Format

In order to save a sound file in either ADPCM (4:1) or SDXC (2:1) compressed format, you must first load the file.

Note: 3DO no longer supports the SDXC format. Use *Squash Sound* for compression to M2 2.0 formats.

Loading a Sound File

To load a sound file:

1. Open SoundHack and wait for the Splash screen to disappear.
2. Use the file selection dialog to choose a sound file to open.

The Soundfile Information window appears, showing the file name, sample rate, length in seconds, number of channels, type and numeric format of the sound file.

The Soundfile Information dialog always describes the "current" sound file. Only one input sound file may be opened at a time, so any command selected from one of the other menus always applies to the current sound file. To change the current sound file, close it and open a new one.

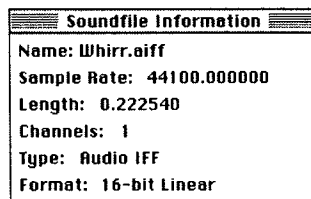


Figure 8-1 Soundfile Information window.

Note: The Open command opens the following types of sound files: Sound Designer II (Sd2f), DSP Designer (DSPs), MacMix (MSND), IRCAM/BICSF (IRCM), NeXT/Sun (NxTs), Audiomedia (Sd2f), Audio Interchange File Format (AIFF), Audio Interchange File Format-C (AIFC), and Microsoft WAVE (RIFF).

Playing the Current Sound File

SoundHack plays only files in AIFF format. If the file you open is in a different format, use the Save A Copy command described below to change the format to AIFF. Once the current sound file is in the correct format, follow these steps to play it:

1. From the File menu, choose lisTen to AIFF file—or type Command T.
The sound plays on your Macintosh.
2. Click the mouse to stop the file playing.

Converting the Current Sound File to Compressed Format

SoundHack lets you load and save sound files in a variety of formats. To convert a file to a compressed format that you can play on the current 3DO system, see the description of Squash Sound.

Reading/Writing Raw Files

Another use of the Save A Copy command's Output Soundfile Format dialog is to create headerless or raw files. Within your 3DO program code you can read in a raw file that SoundHack recognizes as headerless data, saving a small amount of memory.

You'd also want to create a headerless file if you have all the sample information set up in your program with a `CreateSample()` call, and you want to load in the raw data to play it. If there is an AIFF header in the file, the header would be played as garbage.

To make a file headerless;

1. Choose Save A Copy from the File menu.
2. In the Output Soundfile Format dialog that appears, select Headerless Data from the File Type pop-up, as shown in Figure 8-2 below.
3. Once you have a raw file, you can use the Open Any command to load it into SoundHack.

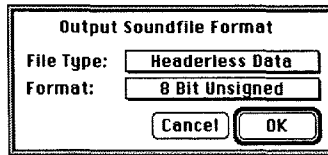


Figure 8-2 *Output Soundfile Format dialog for changing to Headerless Data file type.*

Changing Header Information

SoundHack offers two options for changing a sound file's header information, each of which affects different parameters, as described below.

Header Change Command

Use this command from the Hack menu to change the sample rate, number of channels, and data format of the current open file. If you open a headerless file, use this dialog to set file information before saving a copy.

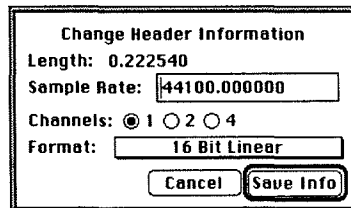


Figure 8-3 *Header change dialog.*

- ◆ To change the sample rate of a sound file, enter a value in the Sample Rate box. Lower sample rates reduce the size of the sample, but trade off audio quality. Note that sample rates below 11 KHz (telephone quality) are not recommended.
- ◆ Click one of the channel selection buttons to change the number of channels (i.e., mono or stereo) you want the file divided into.
- ◆ Change file format in the Format pop-up. Currently only 8- and 16-bit linear formats are available. Selecting the 8-bit format reduces sample size by half, but trades off audio quality.

Loops & Markers Command

Use the Loops & Markers command in the Hack menu to modify the current sound file's header information, if there is any. This command works with AIFF, AIFC, and Sound Designer II formats. The dialog lets you:

- ◆ Move the markers in the header so that they line up on quad-bytes. Note that if there are no markers in the file, you cannot add them with this command.
- ◆ Modify a loop in the sound file.
- ◆ Modify information in the header of a MIDI file that lets you use a single MIDI channel for different sounds, not just different pitches.
- ◆ Change AIFF/AIFC-specific information (i.e., the low-, base-, and high-note numbers).

Markers, Loop Points, Etc.

☐ ☐ Marker #: No Markers

Marker Time:

☐ ☐ Loop #: No Loops

Loop Start Time:

Loop End Time:

Below values for AIFF & AIFC only.

Low note #: Base note #: High note #:

0 60 127

Detune: Gain:

0 0

High Velocity: Low Velocity:

127 1

Cancel Change

Figure 8-4 *Loops and Markers dialog.*

SoundHack User's Guide

Introduction

This chapter discusses all the functionality of SoundHack not covered in Chapter 8, "Essential Features of SoundHack." If you need more information, several dialogs have a Help button to click for information about the process they invoke.

Keep in mind that sound file processing options in the Hack menu are complicated and time consuming, and also tend to take over your Macintosh. If, for example, you're using Phase Vocoder to change either the pitch or length of a sound file and wondering if your Macintosh has frozen up, it probably hasn't.

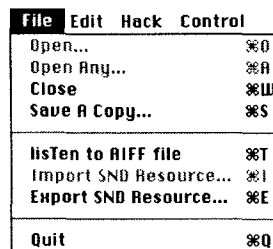
Chapter Overview

The following information appears in this chapter.

Menu	Page
The File Menu	54
The Edit Menu	56
The Hack Menu	56
The Control Menu	70
Bibliography	71

The File Menu

The File menu is concerned with basic sound file management. Figure 9-1 below shows the SoundHack File menu.

A screenshot of the 'File' menu from the SoundHack application. The menu is displayed as a vertical list of options. The first four options are grouped together: 'Open...', 'Open Any...', 'Close', and 'Save A Copy...'. The next three options are separated by a horizontal line: 'lisTen to AIFF file', 'Import SND Resource...', and 'Export SND Resource...'. The final option is 'Quit'. Each option has a keyboard shortcut indicated by a small icon (a square with a diagonal line) and a letter or symbol to its right.

File	Edit	Hack	Control
Open...			⌘O
Open Any...			⌘A
Close			⌘W
Save A Copy...			⌘S
<hr/>			
lisTen to AIFF file			⌘T
Import SND Resource...			⌘I
Export SND Resource...			⌘E
<hr/>			
Quit			⌘Q

Figure 9-1 *File menu.*

Open Command

Opens the following types of sound files:

- ◆ Sound Designer II (Sd2f)
- ◆ DSP Designer (DSPs)
- ◆ MacMix (MSND)
- ◆ IRCAM/BICSF (IRCM)
- ◆ NeXT/Sun (NxTs)
- ◆ Audiomedia (Sd2f)
- ◆ Audio Interchange File Format (AIFF)
- ◆ Audio Interchange File Format - C (AIFC)
- ◆ Microsoft WAVE (RIFF)

If the file doesn't have a Macintosh four-character file type, it appears in the Open File dialog box if it has one of these extensions:

- ◆ .aifc
- ◆ .aiff
- ◆ .au
- ◆ .irc
- ◆ .sf
- ◆ .snd
- ◆ .wav

Once the file is open the Soundfile Information dialog box appears (see Figure 9-2.) This dialog gives the name, sample rate, length in seconds, number of channels, type, and numeric format of the sound file.

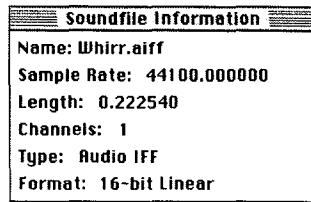


Figure 9-2 *Soundfile Information dialog.*

If you select a process from the Hack menu, SoundHack processes the file displayed in the Soundfile Information box. Only one input sound file may be open at a time.

Open Any Command

Opens any file as a sound file. This command is useful for opening headerless and text sound files. Text sound files should be formatted so that each line is a fixed point sample. Here is an example of how the text should look:

Example 9-1 *Text sound file example.*

```
-0.054688  
-0.015625  
-0.007812  
0.015625  
0.078125  
0.093750  
0.117188  
0.093750  
0.015625  
0.000000  
-0.117188
```

Headerless files (both text and raw) must be saved to another format before being processed.

Close Command

Closes the currently open sound file. You must close the current sound file before opening another one. Save A Copy Command, because the Open command in the File menu remains dimmed until you close the current file. You can, however, open a new file in the Finder (if it is a SoundHack file); this automatically closes the current file.

Save A Copy Command

Saves a copy of the open sound file in any sound file format. This command makes it possible to convert sound files from a variety of formats to AIFF files. Save a Copy offers two compressed 3DO file formats. To access them, select first the File Type Audio IFC in the Format pop-up, then 8 Bit 2:1 3DO SQXD or 4 Bit 4:1 ADPCM in the dialog below.

Note: *This is the primary command to use if your main goal is file conversion.*

lisTen to AIFF File command

Plays the open AIFF file through the Macintosh speaker. You need SoundManager 3.0 or an AV machine for this to work properly. (AV Macintoshes all have SoundManager 3.0 installed.)

Import SND Resource Command

Lets you convert an Apple sound resource to a sound file. This command does not work with compressed .snd resources.

Export SND Resource Command

Lets you write part of a sound file into an Apple sound resource. The length of the sound resource is limited by the amount of memory allocated to SoundHack.

Quit Command

Quits the application.

The Edit Menu

The commands in this menu are not relevant in this application.

The Hack Menu

Commands in the Hack menu (shown in Figure 9-3) are responsible for all sound file processing. Keep in mind that sound file processing options in the Hack menu are complicated and time consuming to execute, and also tend to take over your Macintosh. If, for example, you're using Phase Vocoder to change either the pitch or length of a sound file and wondering if your Macintosh has frozen up, it probably hasn't. See Chapter 8, "Essential Features of SoundHack," for information on the first two commands in the Hack menu: "Header Change Command" on page 50, and "Loops & Markers Command" on page 51.

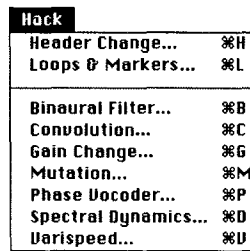


Figure 9-3 Hack menu.

Note: Since SoundHack uses most of the available memory, your Macintosh may appear to be frozen while processing. Have patience.

Binaural Filter

SoundHack has a binaural filter, which processes a monaural sound file so that the resulting file puts the signal at a simulated position around the listener's head, emulating stereo. SoundHack does this by using an HRTF (head-related transfer function) as a filter, with a function for each position around the head.

Binaural spatialization requires stereo headphones to perceive the difference between monaural sound and stereo achieved with a binaural filter.

Selecting this command brings up the dialog shown in Figure 9-4.

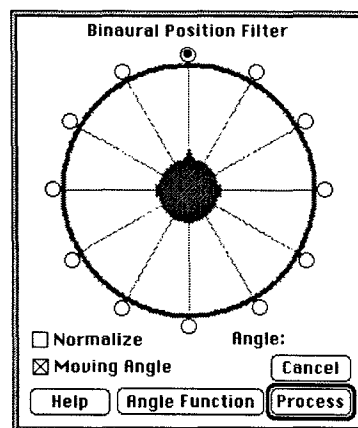


Figure 9-4 Binaural Position Filter dialog.

Processing a File With the Binaural Filter

1. Choose Binaural Filter from the Hack menu.
2. In the dialog that appears, click the position you want the sound to originate from or enter an angle.

The head in the dialog is looking frontward, so the currently selected position in Figure 9-4 would, for example, play the sound 60 degrees left of front.

Using the Binaural Filter

To use a binaural filter, follow these steps:

1. Enter the desired position in the Angle dialog box (in degrees) *or* click one of the radio buttons on the outside of the position wheel.

The position wheel shows the head of the listener, nose pointing upward. Select the position based on that orientation.

Note: *SoundHack has HRTF filter functions for 12 positions around the head of the listener, but if you enter an angle value between two positions, SoundHack mixes the two filters on both sides of the selected value.*

2. To normalize the output after computation—that is, bring it to the loudest possible level—check the Normalize box.
3. To do moving spatialization, check the Moving Angle box. This brings up an Angle Function button; clicking the button brings up an edit dialog that functions as described in “Using the Edit Function Dialog” on page 65.

Unfortunately, because of the limits of the present binaural filters, the moving spatialization effect is less convincing than the stationary effect.

4. Click Process.

For More Information

The binaural filters are sample rate conversions (for 44,100 samples per second) of those used in the dissertation “Control of Auditory Distance” by Durand Begault; those conversions are approximations of the averaged monaural transfer functions shown in Blauert’s *Spatial Hearing*. See the bibliography at the end of this chapter.

Convolution Command

Convolution is the process of multiplying the spectra of two sound files, an input file and an impulse response file, creating a type of cross-synthesis in which common frequencies are reinforced. The sound is processed block by block, with each block as large as the impulse response.

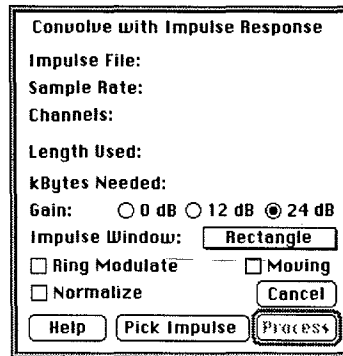


Figure 9-5 Convolution dialog.

Table 9-1 describes Convolution dialog options.

Table 9-1 Convolution dialog options.

Option	Description
Length Used	Specifies how much of the impulse response file to use.
Kilobytes Needed To Process File	Estimate of the application memory size to set for processing. For large impulse responses—above the default of 900—you must quit SoundHack and reset application memory size.
Filter Gain (Gain)	SoundHack attempts to automatically scale the amplitude of filter gain, but this value is impossible to predict. The Filter Gain buttons control the amplitude of the filter gain. Select 24 dB for most cases. Select one of the other options to avoid clipping if the input and impulse response have similar spectra, since extreme resonances will occur. If a clipped output still seems unavoidable, save the output in NeXT floating point format, then use the Gain Change command from the Hack menu to normalize it back to an integer format.
Impulse Window	Applies a selected envelope to the impulse before convolution, resulting in smoother convolution. Desirable for a moving impulse response convolution since the impulse response changes for every block of samples processed. Especially true for a moving ring modulation. A triangular window is best for smoothing; a rectangular window has no effect.
Ring Modulate	Performs a ring modulation—or convolution in frequency—between the two sound files being processed.

Table 9-1 Convolution dialog options. (Continued)

Option	Description
Normalize	Normalizes output back to integer format after computation.
Moving	<p>Performs a moving impulse response convolution: A window moves through the impulse response file, selecting a new impulse response after every block of processing.</p> <p>Set window size in the Length Used field. The window moves through the impulse response file at a rate that ensures that the ends of the impulse response file and the input file are reached simultaneously. If the impulse response file is longer than the sound file, sections of the impulse response file are skipped. It is a good idea to set Filter Gain to Low if using the moving impulse—or to save the file in a floating point format—because the scaling is fairly unpredictable.</p>

Example for Moving Input Convolution

With 10-second input file, a 5-second impulse response file, and Length Used set to 1.0 for 1-second impulse response windows, the process looks like Figure 9-6:

Input File 10 seconds long

A	B	C	D	E	F	G	H	I	J
---	---	---	---	---	---	---	---	---	---

Impulse Response File 5 seconds long

a	b	c	d	e	f	g	h	i	j
---	---	---	---	---	---	---	---	---	---

Output File

A*ab	B*bc	C*cd	D*de	E*ef	F*fg	G*gh	H*hi	I*ij	J*j0
------	------	------	------	------	------	------	------	------	------

Figure 9-6 Example for moving convolution.

- ◆ The first 1.0-second frame of the input file (A) is convolved with the first 1.0-second frame of the impulse response file (ab).
- ◆ The window on the input file is moved 1.0 second forward to B, but the window on the impulse response file is moved only 0.5 seconds to bc. Both files finish at the same time. Actually, the impulse response file reaches the end first and the last impulse response is zero-padded.

Gain Change Command

The Gain Change command allows scaling the gain/amplitude of a sound file.

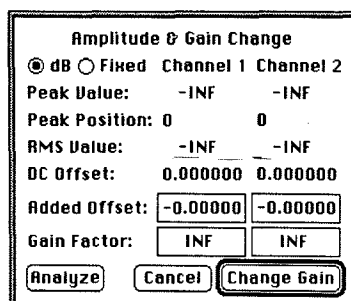


Figure 9-7 Gain Change dialog.

Equivalent functions in Sound Designer II and Alchemy are faster, and you may prefer to use them. However, SoundHack provides an RMS (root-mean-square, or the square root of the sum of the squares of gain/amplitude values, divided by the number of samples) value for the file, and allows a different gain factor for each channel. It also works on floating point and μ Law files.

Use Gain Change to do the following:

- ◆ Click on Analyze to calculate the peak value, peak position (in samples), RMS values, and DC offset. The gain factors are set to normalize both channels independently, and the Added Offsets (a number between 1.0 and -1.0) will be set to correct the file.
- ◆ Click on Change Gain to create a new file adjusted by the gain factors set. In a monaural file, only the channel 1 information is applicable.

Mutation Command

Mutation takes a source and a target sound file and combines the phase/amplitude pair of each frequency band in these files to produce a "mutant" file. The Spectral Mutation dialog, shown in Figure 9-8, supplies the functions needed to perform the mutation operations.

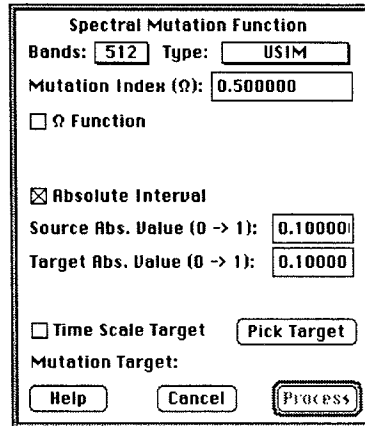


Figure 9-8 *Spectral Mutation dialog.*

Table 9-2 describes the Spectral Mutation Function dialog options.

Table 9-2 *Spectral Mutation Function dialog options.*

Option	Description
Bands	The number of frequency bands.
Type	Pop-up menu of seven spectral mutation functions that produce different types of timbral cross-fades.
Mutation Index (Ω)	Determines the amount of spectral mix, from 0 to 1, between the source and target files that produce the mutant. Ω = 0 results in all source file, Ω = 1 in all target. Ω may vary over the course of the mutation; this produces more dynamic sounds.
Absolute Interval	The mutation functions use two methods to compute intervals between frequency bands: Absolute and Relative. To compute Relative intervals, uncheck the Absolute Interval box. The best way to learn about this function is to experiment with checking and unchecking the box with different spectral mutation types selected. For example, Relative Interval mutations tend to “drift,” and the ISIM mutation type may never “arrive”; this can produce very interesting sonic results.

Option	Description
Ω Function	If the mutation uses Relative Intervals, check Ω Function to set a value for Delta Emphasis, which lets you control the degree to which successive mutation intervals are emphasized in the resulting mutant.
Band Persist	This function is for irregular mutations only (LCM, ISIM, IUIM) and the concatenations. High values for Band Persist (toward 1.0) produce more "stable" mutations; low values (toward 0.0) introduce a kind of frequency pumping at the frame rate. For unusual results, try changing the number of bands.
Time Scale Target	The form of mutation functions in SoundHack require that each sound file (source, target, mutant) be of equal length. The default technique is to truncate the longer of the two files, producing a mutant the length of the shorter file. If Time Scale Target is checked, the target sound file is time-stretched or time-compressed to be of the same length as the source.

Phase Vocoder Command

This process lets you change pitch without changing the length of the sound file, or change length without changing pitch.

SoundHack extracts amplitude and phase information for 16–8192 frequency bands with a bank of filters. If time stretching is desired, these phase and amplitude envelopes are lengthened (or shortened, for time compression), and then given to a bank of oscillators with frequencies corresponding to the filters. For pitch shifting, the envelopes are untouched, and are given to a bank of oscillators with frequencies related by the pitch ratio.

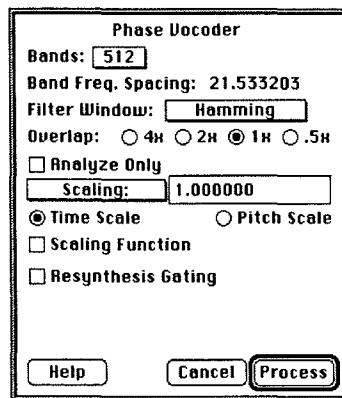


Figure 9-9 *Phase Vocoder dialog.*

Table 9-3 lists all Phase Vocoder dialog options.

Table 9-3 *Phase Vocoder dialog options.*

Option	Description
Bands	Set Bands to the number of filter-oscillator pairs to use. A large number of bands gives better frequency resolution; a small number gives better time resolution.
Filter Window	Lets you choose different pre-FFT (Fast Fourier Transform) windows for different filtering characteristics. Note that only the Hamming, Hanning, and Kaiser give good results.

Table 9-3 *Phase Vocoder dialog options. (Continued)*

Option	Description
Overlap	Adjusts the size of the filter window (relative to the number of filter bands) for analysis and synthesis, and thus, the sharpness of the filter. A large setting (4x) gives the sharpest filter. A sharper filter differentiates better between frequencies that are between bands, but responds more slowly to amplitude changes.
Scaling	Type the scale factor in the Scaling box. Click on the scaling pop-up menu to specify time scaling by the length desired, or pitch scaling by equal-tempered semitones.
Time Scale	Click the Time Scale button if you want time scaling.
Pitch Scale	Click the Pitch Scale button for pitch scaling.
Analyze Only	Creates a Csound-compatible pvoc analysis files.
Scaling Function	To change the time expansion factor or the pitch transposition factor during processing, click the Scaling Function box, then select the Edit Function button that appears. Using the Edit Function dialog is described below.
Resynthesis Gating	Resynthesis gating refers to how much of a sound the Phase Vocoder reconstitutes. It works by breaking a sound down into component sine waves, altering the sine waves in particular ways, and recombining them into a new sound. This is called "resynthesis." Checking Resynthesis Gating lets you set Minimum Amplitude and amplitude Threshold Under Maximum, to tell the Phase Vocoder not to try to resynthesize parts of the sound that would be inaudible in the final product anyway. This way processing takes less time because it skips resynthesizing sounds below the amplitude you specify.

Using the Edit Function Dialog

Use the Edit Function dialog to create control functions. You can draw in the function window; set upper and lower limits for the function; clear, invert, reverse, smooth, or shift the function; or use simple waveforms (up to 100 cycles) as a control function. There is no facility for selecting, copying, pasting, or cutting.

If the Time Scale button is turned on in the Phase Vocoder dialog, the Edit Function dialog shows the range of the vertical axis in the top-left box and of the horizontal axis in the bottom-left. With the Pitch Scale button on, the boxes show

the maximum and minimum pitch range. The Time Scale Edit Function dialog indicates a wave's time scale; the Pitch Scale Edit Function dialog indicates semitones.

The set of boxes in the upper-left side of the window represents default scaling functions, which can be modified by drawing or clicking with the mouse in the editing window. Also, double-clicking on a function applies it to what's currently in the editing window.

The command buttons below these boxes offer the following functions:

- ◆ **Clear**—Returns to the default, a flat horizontal line
- ◆ **Reverse**—Flips the waveform horizontally
- ◆ **Invert**—Flips the waveform vertically
- ◆ **Shift**—Executes a phase shift according to a number (0–399) you specify
- ◆ **Smooth**—Smooths the waveform very slightly

After making selections and adjustments in the Edit Function dialog, click Done and then click Process in the Phase Vocoder dialog. An Output Waveform dialog shows the wave being processed. When processing stops you can play the sound.

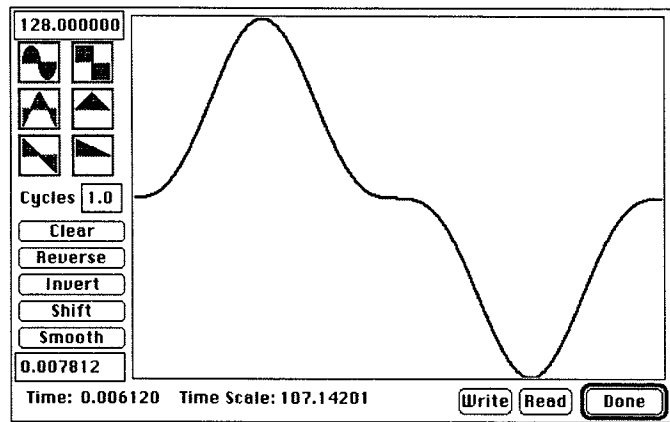


Figure 9-10 *Edit Function dialog.*

Spectral Dynamics Command

The Spectral Dynamics command performs standard dynamics processing (gating, ducking, expansion, compression) on each spectral band individually. It has individual threshold detection for each band, so that the dynamics process can be active in one band while inactive in another.

The process can be limited to affect only a specific frequency range by use of a threshold. You can decide to affect only sounds that are above the threshold or sounds that are below it. The threshold level can be set to one value for all bands, or it to a different value for each band by reading in and analyzing a sound file. This sound file's amplitude spectrum is used for the thresholds for each band. This is especially useful if there is a sound you want to emphasize or deemphasize.

When you choose the command, the Spectral Dynamics Processor dialog appears:

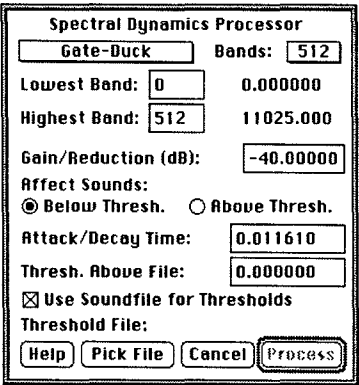


Figure 9-11 Spectral Dynamics Processor dialog.

The Spectral Dynamics Processor dialog offers the following options:.

Table 9-4 Spectral Dynamics Processor dialog options.

Option	Description
Gate-Duck pop-up	Lets you select the type of process to use; i.e, gating, ducking, expansion, or compression.

Gate-Duck

Gate-Duck

Expand

Compress

0

512

Table 9-4 *Spectral Dynamics Processor dialog options.*

Option	Description
Bands pop-up	Sets the number of filter bands into which to separate the sound. 512 is a good compromise for the number of bands at a 44,100 sample rate as each band is about 43 Hz apart and the filters used have a $(512 \times 2) / 44,100$ - or .023-second delay. This is a pretty good frequency resolution (if no partials are closer than 43 Hz) without too much time smearing.
Highest Band / Lowest Band	Limits the frequency range affected.
Gain/Reduction	Sets the amount of gain or reduction for the bands that are past the threshold. For compression and expansion, this box becomes the ratio. The compressor and expander hold the highest level steady and affect lower levels (also known as “downward” expansion or compression).
Attack/Decay Time	When using a duck or gate, the amount of time set here is deducted before the point at which the sound goes above the threshold you set, and added after the point at which the sound falls below the threshold.
Threshold Level	Sets the threshold level.

Varispeed Command

As a sample rate converter, SoundHack is slower and possibly less accurate than Sound Designer II or Alchemy, so this function may be superfluous to users who own that software. What SoundHack has, however, is a variable sample rate conversion utility (varispeed).

Choose Varispeed under the Hack menu and check Varispeed in the dialog that appears. This brings up the Edit Function dialog that also appears when Pitch Scale is turned on in the Phase Vocoder dialog (described above), and lets you manipulate a waveform to vary the rate over the length of the sound sample in the same way you modified the pitch scale with Phase Vocoder.

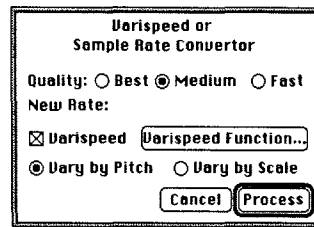


Figure 9-12 Varispeed dialog.

Table 9-5 lists all Varispeed dialog options.

Table 9-5 Varispeed dialog options.

Option	Description
Quality	The Quality buttons control the size of the smoothing filter used, and the resultant quality of interpolation/decimation.
Varispeed box	Click the Varispeed box to enable the Varispeed feature.
Varispeed Function	Clicking the Varispeed Function button displays the Edit Function dialog, giving you control over a 10-octave varispeed.
Vary by Pitch/Vary by Scale	The Vary by Pitch and Vary by Scale buttons allow you to draw a curve for either pitch or scaling factor.

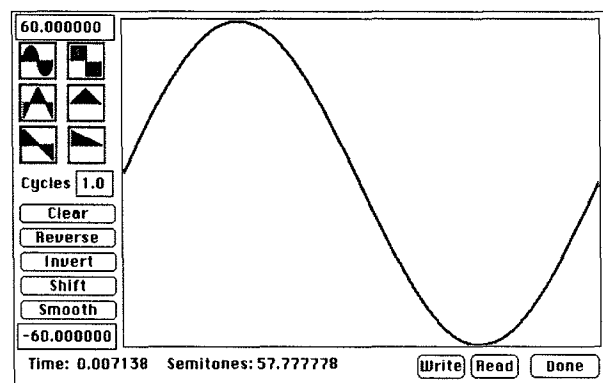
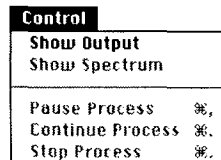


Figure 9-13 Varispeed Edit Function dialog.

The Control Menu

The Control menu lets you display the Output window and pause, continue, or stop a process.

A screenshot of a Macintosh menu titled "Control". The menu is open, showing several options. The first two options, "Show Output" and "Show Spectrum", are highlighted with a light gray background. Below these are three options: "Pause Process", "Continue Process", and "Stop Process", each followed by a keyboard shortcut symbol (⌘, ⌘, and ⌘ respectively).

Control	
Show Output	
Show Spectrum	
Pause Process	⌘
Continue Process	⌘
Stop Process	⌘

Figure 9-14 Control menu.

Show Output Command

Selecting this command displays a window to show your sound whenever SoundHack writes sound to your output sound file, except during file copying and normalization. This slows down processing somewhat.

Show Spectrum Command

Selecting this command displays a window that shows the spectrum of the sound file as it's being generated.

Pause Process Command

This command allows you to pause during a long process in order to use your Macintosh without starting the processing over. If you are running a convolution, it may take a while—up to three minutes—to pause.

Continue Process Command

Resumes processing where you left off.

Stop Process Command

Kills the currently running process and closes the output sound file.

Bibliography

For additional reading on sound design:

Begault, Durand, *Control of Auditory Distance*. Doctoral dissertation, University of California.

Blauert, J., *Spatial Hearing*. Cambridge, MIT Press, 1983.

Dolson, Mark, "The Phase Vocoder: A Tutorial." *Computer Music Journal* 10:4, 1986.

Moore, R. Richard, *Elements of Computer Music*. Englewood Cliffs, NJ, Prentice Hall, 1990.

Note: *Selecting the Bibliography command from the Apple menu will show you additional references.*

AudioThing

Introduction

The AudioThing is an audio preview tool that is a Macintosh Sound Manager extension. With the introduction of Sound Manager 3.0, Apple provided developers with a way to extend the audio capabilities of the Macintosh by adding new hardware, compression schemes, and processing effects. Sound Manager 3.0 was built on the same modular architecture that QuickTime uses.

AudioThing is a sound-decompression component that allows the Sound Manager to play back audio data that is compressed using one of the 3DO-supported methods, such as ADPCM 4:1 or CBD2. This data can be in AIFC files, or because of the modular nature of the Sound Manager and QuickTime, the sound data can be in the audio track of a QuickTime movie. It enables any application that uses standard Sound Manager or QuickTime calls to produce audio.

System Requirements and Options

AudioThing is a fat-binary component. It requires the following software:

- ◆ SoundManager 3.0.

Note: *SoundManager 3.0 is not PPC Native. Sound Manager 3.1 is native, so Power Macintosh users may need to use Sound Manager 3.1. Power Macs with SM3.0 need to be tested.*

- ◆ An AIFF sample player that uses the Sound Manager for AIFC file playback, or a QuickTime movie player that supports AIFF import or standard Translation Manager conversions.

Installation

To install AudioThing, simply place it in your Extensions folder. The next time you reboot, it will be installed and ready to use.

Using AudioThing

This section steps you through the use of AudioThing.

1. Using SquashSnd, or any other 3DO Tools that produce compressed audio, create an AIFC file compressed with one of the following methods:
 - ◆ ADP4 - ADPLM 4:1
 - ◆ SQS2 - 2:1 M2 native
 - ◆ CBD2 - 2:1; good for stereo
2. Using any application that can play AIFF files (a small application is provided with AudioThing that does this), open and play back your audio file. The sound you hear, decompressed in real-time, is what your file will sound like when played on the 3DO hardware.
3. (This step is optional.) Using MoviePlayer 2.0 or any application that supports QuickTime Data Exchange, import your sound into a new or existing movie. During playback, QuickTime uses the AudioThing to decompress the audio. When combined with video compressed with EZFlx, or MPEG, you should have a fairly realistic preview of your content as it will appear on the 3DO.

Tips, Tricks, and Troubleshooting

Available audio tools

The 3DO Company provides developers with several different audio tools:

SoundHack

SoundHack provides, among other things:

- ◆ AIFF-C format files
- ◆ 3DO 2:1 compression
- ◆ ADPCM 4:1 compression in Intel/DVI format
- ◆ decryption of 3DO Content Library sound files
- ◆ dialog box editing of the AIFF instrument and marker chunks
- ◆ sample rate conversion (for example 44.1 kHz to 22.05 kHz)
- ◆ time stretching and pitch adjustments
- ◆ binaural filter spatialization
- ◆ convolution between two samples

SquashSound

SquashSound is an MPW tool that compresses sound files using the 2:1 3DO formats, and ADPCM 4:1. It uses the same ADPCM algorithm as SoundHack but is useful for batch processing files.

ARIA

The ARIA tool (Assembler for Real-time Interactive Audio) lets nonprogrammers create audio content, that is, sound effects and music, for the 3DO Station. You can create and use MIDI Scores using an interactive graphic user-interface and hear the result in real time on the 3DO Station. Specifically, you can:

- ◆ Create music in real time by connecting your favorite MIDI Manager compatible sequencer to the ARIA tool and hearing your music on the 3DO

platform while composing it. In effect, you can turn your 3DO Station into a MIDI synthesizer.

MakePatch

MakePatch is a program that runs on the 3DO development system. It lets you construct custom DSP patche files using a simple script language. The resulting patches may be used like any other DSP instrument template.

AIFF Files

AIFF files in Examples folder

The AIFF sample file in the Example folder were generated with a DigiDesign AudioMedia II board. Just save the files in AIFF mono or stereo file format. Mono is typically used for sound effects. Stereo is typically used for sound tracks. Any software that can save a sound file in AIFF format is acceptable.

AIFF and compression

AIFF is not a compressed file format. The AIFF-C file format supports compression but does not require it. AIFF-C files may have a compression type of NONE, which makes them functionally equivalent to AIFF files. The audio folio supports a 2:1 compression scheme that writes AIFF-C files.

You can create a compressed AIFF-C file using the SquashSound MPW tool or the SoundHack tool. SquashSnd 2.2 can compress both mono and stereo AIFF and AIFF-C files and do 2:1 sample rate conversion.

Preparing Audio

The sections below briefly discuss how to prepare your audio output for mastering and editing.

Mastering audio

To master audio, record at 44.1 kHz and use a DAT machine with digital I/O capabilities, like the Panasonic SV 3700. Then transfer your recordings digitally to the Macintosh. Sound files and samples should be normalized to maximum amplitude to take full advantage of the 16-bit dynamic range.

Editing Synthesized Audio

The AudioMedia system from DigiDesign is highly recommended for editing synthesized audio. It supports a variety of sample formats, has simple but powerful editing capabilities, and can record or play back digital or analog sound.

SoundEdit Pro from Macromedia works with all types of AIFF files and performs best with 8-bit, 22 KHz files.

SoundEdit uses different algorithms to resample a file and to cut and paste between different sample rate files. If you resample a file to a different bit depth/sampling rate, SoundEdit may add aliasing and degrade the sound.

Use the following method for better results:

1. Create a new empty file.
2. Double-click the bit depth/sample rate icon in the lower-left corner of the new file's window.
3. Change the file's bit depth/sample rate as desired.
4. Return to the original file.
5. Select all.
6. Copy.
7. Go to the new file.
8. Paste.

For sample rate conversion, use SoundHack or SoundDesigner. An early bug in SoundHack that caused a pop at the beginning has been fixed.

AIFF Samples

Overview

This appendix provides some information about the AIFF samples available on the 3DO Toolkit CD-ROM; discussing the following topics:

Topic	Page
File names	80
Auditioning samples from the Sound Designer II Edit window	80
Folder contents	81

File names

All samples in this collection conform to the following name convention:

Name.Description.aiff

For example: *Piano.A5LM44k.aiff*

The description part of the filename can include the following:

Name Element	Discussion
Note Name (if applicable)	Based on middle C = C4 (not the Yamaha C3) s = sharp (no flats used)
Loop Type(s)	L = sustain loop LR = release and sustain loop
Mono/Stereo	M = mono S = stereo
Sample Rate	44k (44,100 kHz) 22k (22,050 kHz) (not Mac 22k) Vk (variable rate)

Auditioning samples from the Sound Designer II Edit window

To prevent audio aliasing, Digidesign put a software filter in their Sound Designer II digital audio program. As a result, you will hear pops even when auditioning clean 22 kHz samples from the Sound Designer II Edit window. (It also makes 22 kHz samples sound more boxy than they really are.)

All the 22 kHz samples in this collection are pop-free. To hear them correctly, use the Open dialog box and click on the Play icon. This bypasses the software filter.

Also, the Apple Macintosh™ does not produce sound resource files at true 22,050 Hz but instead at 22,100 kHz. Some 3DO developers have noted lip synch problems because the Mac sample plays at 22,100 Hz while the 3DO DSP instruments always play at true 22 kHz (22,050 Hz).

To save RAM, 3DO developers might consider using the few “Vk” samples that are provided. These samples were recorded at SRs appropriate to their pitch (i.e., at the best Nyquist frequency). Vk, and 22k samples save RAM, but 3DO developers must take a DSP bandwidth hit to use them because the variable rate DSP instruments take up more code space and ticks.

Folder contents

The samples in this collection are divided into five main folders:

- ◆ GMPercussion22k
- ◆ GMPercussion44k
- ◆ PitchedL
- ◆ PitchedLR
- ◆ Unpitched

Most 3DO developers need only the samples in the first three folders.

Samples in GMPercussion22k and GMPercussion44k

This collection consists of a nearly complete set of samples for a General MIDI drum kit. The only difference between the contents of the two folders is sample rate. Note that the 22 K samples take up half the space of the 44 K samples and in most cases sound about as good.

The General MIDI Percussion support in this 3DO release consists of AIFF files and a PIMap (+GMPercPIMap44k.txt and +GMPercPIMap22k.txt) that corresponds to the Percussion Map of the General MIDI implementation. You should edit copies of those PIMaps that only reference the samples you need. General MIDI assigns percussion sounds to notes 35-81 on MIDI channel 10, as follows:

- 35 Acoustic Bass Drum
- 36 Bass Drum
- 37 Side Stick
- 38 Acoustic Snare
- 39 Hand Clap
- 40 Electric Snare
- 41 Low Floor Tom
- 42 Closed Hi-Hat
- 43 High Floor Tom
- 44 Pedal Hi-Hat
- 45 Low Tom
- 46 Open Hi Hat
- 47 Low-Mid Tom
- 48 Hi-Mid Tom
- 49 Crash Cymbal 1
- 50 High Tom
- 51 Ride Cymbal 1
- 52 Chinese Cymbal
- 53 Ride Bell
- 54 Tambourine
- 55 Splash Cymbal

56 Cowbell
57 Crash Cymbal 2
58 VibraSlap
59 Ride Cymbal 2
60 Hi Bongo
61 Low Bongo
62 Mute Hi Conga
63 Open Hi Conga
64 Low Conga
65 High Timbale
66 Low Timbale
67 High Agogo
68 Low Agogo
69 Cabasa
70 Maracas
71 Short Whistle
72 Long Whistle
73 Short Guiro
74 Long Guiro
75 Claves
76 Hi Wood Block
77 Low Wood Block
78 Mute Cuica
79 Open Cuica
80 Mute Triangle
81 Open Triangle

Multiple sample choices

In this release, multiple sample choices are provided for the following:

- ◆ Acoustic Snare (note 36)
- ◆ Open Hi-Hat (note 46)
- ◆ Crash Cymbal 1 (note 49)
- ◆ Ride Cymbal 1 (note 51)
- ◆ Chinese Cymbal (note 52)
- ◆ Splash Cymbal (note 55)
- ◆ Cowbell (note 56)
- ◆ Low Conga (note 64)
- ◆ High and Low Agogos (notes 67 & 68)

Empty assignment slots

The following assignment slots, for which there are no appropriate audio samples in the 3DO Content Library, have been left blank in the PIMap score file:

- ◆ Hi and Low Bongo (notes 60 & 61)
- ◆ Short and Long Whistle (notes 71 & 72)
- ◆ Low Wood Block (note 77).

Tom-Tom Samples

The tom-tom descriptions (Low Floor Tom, High Floor Tom, Low Tom, Low-Mid Tom, Hi-Mid Tom, and Hi Tom) are roughly matched to their files, but drum aficionados may want to replace one or several of the samples.

Samples in the PitchedL folder

The samples in the PitchedL folder have sustain loop markers and in most cases include sufficient fade outs. Instruments in folders marked "Lite" take up less RAM, but have more abrupt fade outs.

Note: All loop markers have been set on even-numbered samples because the 3DO system reads samples in 4-byte packets. By bracketing even-numbered 16-bit samples, these loop segments conform to the requirements of the 3DO system.

Samples in the PitchedLR folder

The samples in this folder are equivalent to the ones in the PitchedL folder but have release loops. All the instrument folders in this main folder have "RL" to distinguish them from the instrument folders in "PitchedL." The 3DO system makes it possible to use release loops.

Note: You should only use samples from this folder if you are willing to devote the programming time and DSP bandwidth for envelopes.

Samples in this folder have the following characteristics:

- ◆ Two pairs of loop markers, each bracketing the same sample data.
- ◆ Sounds are abruptly truncated; there is no fade out.

When you use a sample with a release loop, an envelope is needed to ramp the sound down while release loop data is playing. Consider letting the attack portion of the envelope come from the soundfile itself by setting the first segment of the envelope generator at maximum volume. The envelope would then shape the sustain and release loop sounds.

Samples in the Unpitched folder

Most of the samples in this folder are represented in the GMPercussion sets.

Suggested MIDI note maps

Note: This list includes only some of the samples in this collection; 3DO developers will undoubtedly decide on their own cross-over points, depending on the nature of their music and RAM constraints

Flute		Harpsichord		Oboe		Recorder		
C#4	48 -> 50	C2	30 -> 40	D4	48 -> 63	Version 1	Version	
F4	51 -> 56	C3	41 -> 48	F4	64 -> 66	SopranoC5	(not used)	70 -> 73
A#4	57 -> 59	E3	49 -> 52	A4	67 -> 69	SopranoG5	71 -> 84	74 -> 84
D#5	60 -> 64	G3	53 -> 60	C5	70 -> 72	SopranoD6	85 -> 90	85 -> 90
G5	65 -> 68	E4	61 -> 66	E5	73 -> 76	Version 1	Version 2	
C6	69 -> 90	G4	67 -> 68	G5	77 -> 80	TenorC4	55 -> 60	55 -> 60
		C5	69 -> 73	B5	81 -> 85	TenorF3	48 -> 54	48 -> 54
		E5	74 -> 81	F6	86 -> 90	TenorG4	61 -> 70	61 -> 69
		C6	82 -> 90					

Solo Strings		String Ensemble (PitchedL)		String Ensemble (PitchedLR)	
		These composite samples layer several samples of the same note on different string instruments (viola, violin, cello).			
CelloD230 -> 39		C2	30 -> 39	C2	30 -> 39
CelloG240 -> 43		A2	40 -> 55	A2	40 -> 49
CelloC344 -> 52		D4	56-> 68	A3	50 -> 59
CelloA353 -> 57		C5	69 -> 76	D4	60-> 68
ViolaD458 -> 64		A5	77 -> 83	C5	69 -> 76
ViolaG465 -> 73		D6	84 -> 90	A5	77 -> 83
ViolaB474 -> 74				D6	84 -> 90
ViolinE575 -> 82					
ViolinA583 -> 90					

Trumpet	Woodwind Quartet
A3 48 -> 58	C2 30 -> 37
D4 59 -> 63	E2 38 -> 43
G4 64 -> 68	A2 44 -> 45
C5 69 -> 73	A#2 46 -> 48
F5 74 -> 78	E3 49 -> 52
A#5 79 -> 86	C4 53 -> 60
D#6 87 -> 90 (can use A#5 for this if D#6 sounds strained)	C#4 61 -> 63
	F4 64 -> 66
	G#4 67 -> 72
	D#5 73 -> 75

The InstrumentSamples folder

The Instrument Samples folder contains AIFF files that 3DO developers can use to attach to MIDI notes in their interactive scores. Samples are presented in various stereo/mono 44k/22k formats. Some have release loops in addition to sustain loops. A list is provided in a separate chapter.

DumplIFF Sample Output

Introduction

See the example CD-RO for a sample of Mof DumpIFF output. The file being dumped is a 3SF file created using MakeScore on the Zap.mf and Zap.pimap files.

This is a hierarchical format in which each element is an IFF Chunk (similar to, but not to be confused with, a DataStream chunk). IFF chunks have a four-character type code, a length, and sometimes a four-character subtype. The subtype is present only for chunks of type FORM, LIST and CAT—the container types. The meaning of the subtype is context dependent: either it identifies types of chunks within it, or more likely it determines specifically what type of chunk this is.

Chunks are listed in the output as the word “chunk” followed by the chunk type, a comment showing where in the file that chunk lives, and the length of the chunk. The contents of the chunk are dumped in some fashion determined by the chunk’s type, surrounded by curly brackets. Container chunks may contain other chunks.

Chunks of type FORM always contain a subtype, so a shorthand is used to identify them: FORM <subtype>.

Below is a brief description of some of the chunks listed in the sample dump:

FORM 3SF — Root chunk of a 3SF file

3VER — 3SF Version chunk

NUMI — Numeric Index chunk. Important chunks are listed in the index for quick access. Each entry contains the unique ID number for that chunk and the offset in the file where that chunk can be found.

LIST SSET — SoundSet chunk. There is one root level SoundSet that collects all of the sounds that are associated together. SoundSets can contain other SoundSets, but usually they contain AXEs and LICKs.

FORM AXE — A logical instrument. This is often an IFF sample.

FORM LICK — A sequence to play on an axe. This may be MIDI data.

FORM MLIK — The Main Lick of a SoundSet. This is played by default.

SHDR — Set header. This contains the name and id number of the SoundSet, and includes a Name Index for each entry in the SoundSet

IHDR — Item header. This contains the name and id number of the following Item in a SoundSet.

NAMI — Name Index. This is an index table that associates an item's name with its offset in the file, for quick location.

PROG — Program chunk. This identifies an instrument to use for this MIDI program and includes a list of sample references.

SMPR — Sample Reference. This chunk contains a reference to a sample by name. The sample may be included in this file, or as a separate file on disk.

Most other chunks are the standard chunks that appear inside AIFF and AIFC files.

3SF File Format Specification

Introduction

This appendix describes the 3SF File format, known as the “3DO Score File 95” standard. The 3SF Format, like the standard Audio-IFF and Audio-IFC formats, conforms to the IFF file standard “EA IFF 85” standard developed by Electronic Arts (EA).

Caution: *DO NOT RELY ON SPECIFIC FILE FORMAT INFORMATION, AS IT WILL INEVITABLY CHANGE WITH SUBSEQUENT RELEASES, ALTHOUGH THE CONCEPTUAL UNDERPINNINGS WILL REMAIN VALID.*

This chapter covers the following topics:

Topic	Page
File Types and Extensions	89
Data Types	90
File Structure	92

File Types and Extensions

A file in 3SF format created on the Macintosh must have a file type of 3SF. On platforms where suffixes are used to determine the type of a file, we recommend that a 3SF file should have the suffix *.3sf* or *.3SF*. Lower case is preferred. Following this convention facilitates file sharing between people using different hardware platforms.

Data Types

Data structures in this document are described in a C-like language. The data types are listed below:

int8

8 bits, signed, two's complement format. An int8 can represent an ASCII character or a numerical value from -128 to 127 (inclusive).

uint8

8 bits, unsigned. Represents a numerical value from 0 to 255 (inclusive).

uint7

Low 7 bits in a byte, unsigned. The high bit of the byte must be zero. Represents a numerical value from 0 to 127 (inclusive).

int16

16 bits, signed, two's complement format. Represents a number from -32768 to 32767 (inclusive).

uint16

16 bits, unsigned. Represents a number from 0 to 65535 (inclusive).

int32

32 bits, signed, two's complement format. Represents a number from -2,147,483,648 to 2,147,483,647 (inclusive).

uint32

32 bits, signed, two's complement format. Represents a number from 0 to 4,294,967,295 (inclusive).

extended

80-bit IEEE Standard 754 floating-point number (Standard Apple Numeric Environment [SANE]) data type "extended."

bigfixed

A 32-bit signed fixed-point value. The high 17 bits represent the integer part, which can range from -65536 to 65535 (inclusive). The low 15 bits represent the fractional part. This type is capable of representing the int16 and the uint16 ranges without loss. It can also represent the 3DO audio folio's 1.15 fixed-point values with a loss only in the low-order bit, which is negligible.

pstring

Pascal-style string. The first byte is an unsigned count of the number of following text characters. The characters follow the count byte, one per byte. If the number of text bytes is even, then a zero pad byte is added. The pad byte is not included in the count byte. The requirement of the pad byte ensures that a pstring (count byte + text + pad, if any) is always even in total byte length.

cstring

C-style, null-terminated string. If the number of text bytes is even, then an extra zero pad byte is added. The requirement of the pad byte ensures that a cstring (text + terminating-null + pad, if any) is always even in total byte length.

cstring31

C-style, null-terminated string with at most 31 non-null characters, null-padded to be exactly 32 bytes in length

ID

32 bits, the concatenation of four printable ASCII characters in the range ' ' (= sp = 0x20) through '-' (= 0x7E). Spaces may not precede other characters, but trailing spaces are allowed. Control characters are forbidden. Upper/lower case is significant. If documentation refers to a type with fewer than four characters (such as 3SF) then trailing spaces are assumed.

OSType

32 bits, the concatenation of any four ASCII characters, as specified in Inside Macintosh. Upper/lower case is significant. Thus an ID is an OSType, but the converse is not true.

versionstamp

16 bits unsigned. All files in 3DO Score File 95 format must have a versionstamp of 1. Future versions of this standard will have a higher value for versionstamp.

constants

A string of digits represents a number in decimal notation, for example 123, 0, 100. To indicate a number in hexadecimal notation, a 0x prefix is used, e.g. 0x0A12, 0x1.

Data Organization

All data is stored in Motorola 68000 format, high byte first. This is also known as "big-endian" format. In the following table, numerals indicate the bit number within the data. Bit 0 is the least significant bit.

```
int8 | 7 0 |
int16 | 15 8 | 7 0 |
```

```
int32 |31 24|23 16|15 8|7 0|
```

File Structure

The following sections discuss elements of the 3DO file structure. **Please note, however, that the 3SF file format is still likely to change. Some features in the architecture are not yet implemented, and the file format may be changed to make these features work better.**

You should not, therefore, write code that manipulates 3SF files directly. You should use the tools supplied (ARIA, MakeScore, DumpIFF, and the PlayScore programmer interface) for this purpose.

Chunk

The "EA IFF 85 Standard for Interchange Format Files" defines an overall structure for storing data in files. 3SF conforms to the EA IFF 85 standard. We proceed to review those portions of the EA IFF 85 standard that are germane to the 3SF standard. For a more complete discussion, please refer to the documents that define the EA Standard "EA IFF 85 Standard for Interchange Format Files" Electronic Arts 1985, and also "A Quick Introduction to IFF" from Electronic Arts.

An EA IFF 85 file is built up from a number of "chunks" of data. Chunks are the building blocks of EA IFF 85 files. A chunk consists of header + data.

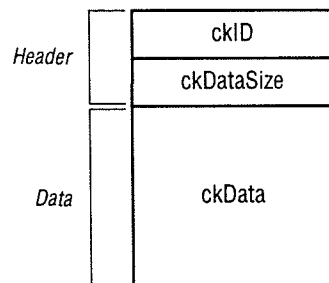


Figure M-1 Basic layout of a chunk.

In 3SF's C-like language, this is described as follows:

```
struct Chunk
{
    ID      ckID;           // chunk ID
    uint32  ckDataSize;     // chunk data size, in bytes
    uint8   ckData[];       // data.
};
```

in which

`ckID;`

Identifies the format of the chunk's data portion. A program can determine how to interpret the chunk data by examining this field.

`ckDataSize`

Specifies the length in bytes of the data portion of the chunk. It does not include the 8-byte header.

`ckData`

Represents the data stored in the chunk. The format is determined by the `ckID`. If the data is an odd number of bytes in length, a zero pad must be added at the end. The pad byte is not included in the `ckDataSize` value, which, in general, can be either odd or even. The pad byte ensures that every chunk begins on an even-numbered byte boundary. The `ckDataSize` thus specifies the "logical" length of the chunk, while the "physical" size of the chunk is rounded up to an even number.

FORM Chunk

A 3SF file is just a single enveloping chunk of type FORM. In general, a FORM chunk's data starts with an ID field, called the "form type," followed by a sequence of one or more chunks. The form type of the enveloping chunk is always "3SF."

The `ckDataSize` of a FORM chunk includes the form type field, and the combined physical sizes of all the contained chunks. Thus the `ckDataSize` of a FORM chunk is always even, because the physical size of each of the nested chunks must be even.

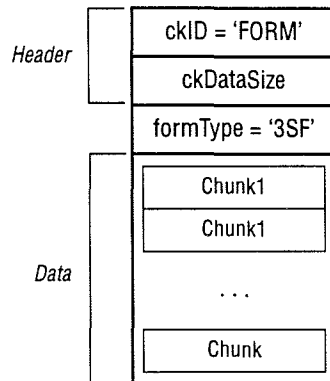


Figure M-2 Structure of the main 3SF FORM chunk.

The pseudo-C for this structure is

```
struct FormChunk
{
    ID ckID;
    uint32 ckDataSize;
    ID formType;
    Chunk chunks[];
};
```

A chunk nested inside a FORM chunk may itself be a FORM chunk, though nested forms (i.e., those other than the top-level enveloping chunk) will have a form type other than "3SF."

Syntax Definitions

The following is a collection of the syntax definitions, adapted from the EA IFF 85 standard. The types Chunk and FORM have been introduced informally above.

```
IFFFile ::= FORM
FORM ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
LIST ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
CAT ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
ContentsType ::= ID-- a hint or an "abstract data type" ID
FormType ::= ID
LocalChunk ::= Property | Chunk
Property ::= Chunk
Chunk ::= ID #{ uint8* } [0]
PROP ::= "PROP" #{ FormType Property* }
```

In this extended Backus-Naur notation, the token “#” represents a count of the following {braced} data bytes. Literal items are shown in “quotes”, [square bracketed items] are optional, and “*” means 0 or more instances. A sometimes-needed pad byte is shown as “[0]”.

File Version Chunk

The file version chunk is a chunk of type ‘FVER’. It must always be present in a 3SF file. The data consists of a versionstamp, which represents the version number of the standard itself. All files in 3DO Score File 95 format, as specified in this document, must have a versionstamp of 0x1.

```
struct VersionChunk
{
    ID ckID = 'FVER';
    uint32 ckDataSize = 2;
    uint16 versionStamp = 0x1;
};
```

The syntax definition is

```
FileVersion ::= "FVER" #{ 3SF_95Stamp }
3SF_95_Stamp ::= 0x1
```

Sound Set

A 3SF file can be thought of as a little file volume. There are containers, called “sound sets,” that are analogous to directories. There are also playable items, called “Score Items,” that are analogous to files. Score Items have two main types: licks and axes. An axe is like an instrument, and a lick is like a tune to be played on it.

Licks have many flavors, including MIDI. Axes can include MIDI set-up data, sample data, DSP instruments, etc.

Each sound set can have a special lick called the main lick. It is legal to play a 3SF file, or an entire sound set, provided a main lick is present: playing the file or set is interpreted to mean playing the main lick. This makes sound sets playable items too.

A sound-set is just a chunk of type ‘LIST’, as specified in the EA IFF 85 standard, with a contents-type ID of ‘SSET’. The syntax definition is:

```
LIST ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
SoundSet ::= "LIST" #{ "SSET" [LickProperties]
    [AxeProperties] SetHeader [MainLick]
    (NonMainLick | Axe)* SoundSet* }
LickProperties ::= "PROP" #{ "LICK" Chunk* }
AxeProperties ::= "PROP" #{ "AXE " Chunk* }
```

The SetHeader, NonMainLick, and MainLick, and Axe chunks are described in later sections.

Observe that a sound set may contain nested sound sets: it is then referred to as a parent sound set. To avoid unnecessary disk seeks at run time, such nested sets must follow all the axe and lick score items that have the same parent.

According to the IFF standard, the property chunks, with an ID of 'PROP', may specify data that is shared by all succeeding chunks of a given type. The 3SF file may contain only PROP chunks for forms of formtype LICK and AXE. The chunks within the PROP chunk include shared audio sample data, etc.

Every 3SF file must have exactly one top-level sound set, along with a file version and a global numeric index. At this stage, we can give the syntax definition for a 3SF file:

```
3SF ::= "FORM" #{ "3SF " FileVersion NumericIndex SoundSet }
```

The smallest possible 3SF file consists of one top-level FORM chunk with one nested FVER chunk, one numeric index chunk, and one SSET containing only a NextItemNumber chunk.

Score Items

A Score Item is either a lick or an axe. A lick normally represents a sequence of sound events, and can be of many types, MIDI being the most usual one. Another way to think of a lick is as a sequence, or as a program for playing sound.

An axe represents an instrument, such as a sound sample, etc. Sound is produced by using an axe to play a lick:

AXE + LICK = SOUND!

Continuing the analogy of licks to program code, think of an axe as data used by the program.

The sound set that immediately contains a score item is called the item's "parent sound set."

Within the 3SF file, any Score Item except a main lick must begin with an item header, which is a chunk of type "IHDR". The header gives the item an "item number," an "item name," and an "item type." The item header is specified in detail in its own section below.

The syntax of a Score Item is therefore given by:

```
Item ::= Lick | Axe
```

Licks

A lick is a FORM chunk with the form-type "LICK" or "MLIK". A so-called "main lick" has the type 'MLIK', and other licks, called "non-main licks," have the type "LICK". Either type of lick normally represents a playable sound, and can be of many types: MIDI, sound sample, etc.

The difference between a non-main lick and a main lick is that

- ◆ A main lick, if present, must be the first lick in its parent sound set.
- ◆ There can be at most one main lick in any sound set.
- ◆ A main lick's name must be "Main."

If a sound set has a main lick, then the sound set itself becomes a playable item: by convention, playing the sound set means playing its unique main lick. If the top-level sound set of a 3SF file has a main lick, then the file itself becomes a playable item.

The syntax of a lick is given by:

```
Lick ::= NonMainLick | MainLick
MainLick ::= "FORM" #{ "MLIK" ItemHeader LickChunk }
NonMainLick ::= "FORM" #{ "LICK" ItemHeader LickChunk }
LickChunk ::= MDLFLick | MDEVLick
              // Only kinds defined so far.
```

LickChunk's chunk ID is always the same as the lick's type as specified in the ItemHeader.

Axes

An axe is a FORM chunk with the form-type "AXE". It was introduced above in the "Score Items" section.

The syntax of an axe is therefore given by:

```
Axe ::= "FORM" #{ "AXE " ItemHeader AxeForm }
AxeForm ::= SampleAxeForm | 3INSAxeForm | MIDIContextAxeForm
SampleAxeForm ::= AIFFAxeForm | AIFCAxeForm
AIFFAxeForm ::= byte image of an AIFF file
AIFCAxeForm ::= byte image of an AIFC file
```

An AxeForm is a FORM. Its form-type is always the same as the axe's type as specified in the ItemHeader. The format for a MIDI Context Axe, and for a 3INS Axe are detailed in a later section.

Item Numbers and Item Names

Item Numbers and Item Names appear in both Set Headers and Item Headers, and they "share the same space." The item name need only be unique among all

score items and sound sets in the parent sound set (if the sound set is not the top-level set). The item number must be unique among all score items and sound sets in the entire file.

The Item Number is a signed four-byte value. The name is a string of undetermined length:

```
ItemName :: cstring
ItemNumber ::= int32
```

The global uniqueness requirement for numbers introduces a difficulty when copying a sound set or score item from one file to another and embedding it in a sound set in the target file.

To resolve these problems, software that transfers a sound sets or score items (called collectively “entities” in the following discussion) from one 3SF file into another should follow these rules and procedures:

- ◆ Only positive Item Numbers may be explicitly assigned by the user (“user” item numbers). The user should not be able explicitly to assign a negative ItemNumber to an entity. By contrast, the software, if it assigns an ItemNumber automatically, must guarantee to assign only a negative ItemNumber, using the NextItemNumber value. (an “automatic” ItemNumber).
- ◆ Only a single entity, or entities that are siblings, should be transferred in a single operation, and they should be transferred to become members of a single sound set in the destination file (the “intended parent”). (Think of dragging files and folders in the Macintosh Finder.)
- ◆ Consider in turn each of the entities being transferred, and to it apply the the subsequent rules in this list.
- ◆ Check for a conflict between the new entity’s ItemNumber in the destination file. If the conflicting ItemNumber in the target file is not one of the new entity’s intended siblings, i.e., is not a current member of the intended parent, and if the ItemNumber is a user number, then the transfer is not allowed: abort the transfer.
- ◆ If there is an ItemNumber conflict and the number is an automatic number, renumber the new entity with a new automatic number (use the nextItemNumber).
- ◆ If there is an ItemNumber conflict between the new entity and an intended sibling, and the number is a user number, offer the user the choice: change old, change new, replace, or cancel. If the user chooses “replace” but the names differ, ask whether to use the old or the new name, or cancel.
- ◆ Having determined in the preceding steps that there is no ItemNumber conflict, there remains the possibility that there is a name conflict between the

new entity and an intended sibling. If so, ask the user whether to replace or cancel.

- ◆ If the user chose "replace" in a preceding step, but now the types differ (sound set, licks, and axes are the three types), warn the user and allow again to cancel.

Set Header

The "Set Header" of a sound set assigns it an `ItemNumber` and an `ItemName`. It also contains a name index for the items directly contained in the set. It is a FORM chunk of contents-type SHDR within the set header (which is an IFF LIST). Its syntax is

```
SetHeader ::= "FORM" #{ "SHDR" SetHeaderChunk }
SetHeaderChunk ::= "SHDR" #{ ItemNumber NameIndex ItemName }
```

Item Header

The "Item Header" of a score item (or a sound set) assigns it a number, a name, and a type. It may contain a dependency list of other items that are used by it. The item header may also specify a list of knobs.

The item type is an ID that specifies the type of item chunks that follow. With Phase I of the 3DO 3SF 95 standard the following are the types that are recognized:

MDFL ("MIDI File")

A lick. After the header, the lick chunk is a MDFL Lick:

```
MDFL Lick ::= "MDFL" #{ Image of type 0 or 1 Midi File }
```

MDEV ("MIDI Event")

A lick. The internal format has not yet been specified (???). After the header, the lick chunk is a MDEV Lick:

```
MDEV Lick ::= "MDEV" #{ Converted data from a Midi File }
```

PIMP ("Processed PIMap, or MIDI Context")

An axe. After the header, the axe chunk is a MIDIContextAxe (defined in a later section).

AIFF

An axe. After the header, the axe form is an AIFF FORM, byte-image of a standard Audio-IFF sound data file.

```
AIFFAxe ::= "FORM" #{ "AIFF" (standard AIFF chunks) }
```

AIFC

An axe. After the header, the axe form is an AIFC FORM, byte-image of a standard Audio-IFC compressed sound data file.

```
AIFCAxe ::= "FORM" #{ "AIFC" (standard AIFC chunks) }
```

3INS

An axe. After the header, the axe form is a 3INS FORM, byte-image of a 3DO system dsp instrument or a 3DO ARIA compiled instrument.

An item header may contain two dependency lists: a load-dependency list and a play-dependency list. Each such list contains a list of references to other items. Dependency lists are discussed in a separate section below.

The item header optionally contains a list of Knobs on the lick. Each knob has a name and a MIDI controller number, in the range 0..127, that it responds to. Certain other options are possible, as described in the "Knobs" section below.

The syntax of an item header is therefore given by:

```
ItemHeader ::= "IHDR" #{ ItemNumber ItemName ItemType [DepsList]
ItemKnob* }
ItemType ::= ID
```

An item header (one with no knobs) can be included in a sound set as well as in a lick. This allows sound sets, as well as score items, to have names and item numbers.

The Global Numeric Index

The global "Numeric Index" is a required chunk, that is, it must always be present in a 3SF file. The numeric index must be kept current by any software that modifies a file.

The Numeric Index begins with a signed 32-bit number that is always negative, which represents the next "automatic" Item Number to be assigned (see Item Numbers and Item Names). This is an architectural requirement to allow numeric keys to remain unique, even between edited versions of a single original file.

Following the Next Item Number value is a list of all `ItemNumber` values for score items and sound sets in a 3SF file, each paired with the file offset of the numbered entity. The list is sorted in increasing order of Item Number values.

Any program that writes a 3SF file must write this chunk. If sound sets or items are added to the file in a way that causes new automatic item numbers to be assigned, the Next Item Number must be used and decremented accordingly.

```

NumericIndex ::= "NUMI" #{ NextItemNumber (ItemNumber Offset)* }
                // sorted
NextItemNumber ::= ItemNumber

```

Specifying a Score Item or a Sound Set

Programs that read and write 3SF files do so through the MakeScore API or the PlayScore API. Both interfaces require the programmer to specify score items and sound sets. As item numbers and item names are unique only in the parent sound set, the 3SF Standard 95 specifies how such references must be resolved.

The first way to specify an item or sound set completely is to use its item number. Using the item number is called a “numeric reference.” Resolve a numeric reference by looking up the name in the global numeric index.

The second way of specifying an item completely is by specifying its parent sound set and its name. Using the name is called a “named reference.” Resolve a named reference by looking up the name in the name index of the parent sound set.

```

CompleteSpec ::=
    NumericRef | ItemNumber | CompleteItemSpec |
    CompleteSetSpec
CompleteSetSpec ::= NumericSetRef | TopSetRef | (ParentSetSpec SetRef)
    // ParentSetSpec must be to a set immediately contained
    // in the sound set specified by ParentSetSpec.
CompleteItemSpec ::= ParentSetSpec ItemRef
    // not used in a file.
    // ItemRef must be to an item immediately contained
    // in the sound set specified by ParentSetSpec.
ParentSetSpec ::= CompleteSetSpec
SetRef ::= Ref :
TopSetRef ::= :
    // TopSetRef is a specification of the top-level sound set.

ItemRef ::= Ref
Ref ::= NamedRef
NumericRef ::= : digit* 0 // colon & numeric cstring
NumericSetRef ::= : digit* : 0 // cstring: colon, number, colon
ItemNumber ::= uint32
NamedRef ::= cstring
    // Colons are illegal. First char must be non-numeric.

```

IMPORTANT RULE: In a 3SF file, an ItemNumber may appear only in the global numeric index (see “The Global Numeric Index” on page 100). Other than there, the only kind of reference or specification that may be saved in a 3SF file is a NamedRef. To avoid creating invalid interdata references when moving

data between 3SF files, you may use the full path names defined above within programs, but you must never save them in the file.

Notice that (1) a set reference ends in a colon, and an item reference does not, (2) a complete specification starts with a colon. This follows the UNIX convention, rather than the Macintosh one. Its advantage is that complete specifications are easily distinguishable from partial ones by the initial colon.

Note further that colons have been chosen to allow a Ref to contain slashes. This means that you can copy a 3DO path name verbatim as a legal Ref, i.e. as the name of an item or set.

Within the file, you can make cross-references without specifying the sound set. This is called an "incomplete specification." An incomplete specification using a named or numeric reference to a score item will be resolved in several stages.

9. The key will be sought in the corresponding index for the parent sound set.
10. If the key is not found in that index, the key will be sought in the parent sound set of the set (the "grandparent" set). This process continues until either the key is found, or the key has been searched for unsuccessfully in the top-level sound set.
11. If the key is not found, and the key is a name, the client program tries to use the item name as a path name to an external file.

This scheme is an "overriding and supplementing" scheme, whereby data with a given name or number that is included in the same sound set is guaranteed to be used in place of data with the same key that is included in an enveloping sound set. It also allows sharing data among many sound sets, by including one copy in an enveloping sound set.

The Name Index

A set header must contain exactly one "Name Index." The name index allows fast lookup of all score items and sound sets immediately contained in the set to which it belongs.

The name index is a pseudo-chunk within the SetHeader chunk. The syntax of the name index is

```
NameIndex ::= "NAMI" #{ (HashedItemName Offset)* } // sorted
HashedItemName ::= cstring // exactly 24 chars, null padded.
```

The named index entries are sorted in increasing order of HashedItemName, as defined by the standard C library call `strcmp()`.

Notice that all entries in a given index have the same length. This, along with the sorting requirement, allows fast searching within the index.

The "hashed item name" consists of a 16 bit hash value followed by an optional ellipsis character and a number of the tail characters of the full item name.

This algorithm is tailored to make it highly unlikely that hashed item names derived from full path names of external files used in 3DO sound development will clash. Such path names are more likely to be unique at the end than at the beginning, and to lie in a few standard directories.

The C++ code for this algorithm is listed here for completeness:

```
typedef uint16 hash_t;
static void Build_CRC_Hash_Table();
typedef uint16 hash_t;
enum { crcBits = 16, crcPoly = 0x1021 };
enum { crcHiBit = 1 << (crcBits-1), crcMask = ((crcHiBit-1)<<1)+1 };
enum { tblBits = 8, tblSize = 1 << tblBits };
static hash_t HashTable[tblSize];
enum { bytesize = 8, shiftAmt = crcBits - bytesize };
//
// Hash helper table creation
//
void Build_CRC_Hash_Table()
{
    hash_t crcPoly_d = crcPoly;
    HashTable[0] = 0;
    for (int d = 0; d<tblBits; ++d) {
        int diff = 1 << d;
        for (int j = 0; j < diff; ++j)
            HashTable[j+diff] = HashTable[j] ^ crcPoly_d;
        Boolean hasHiBit = crcPoly_d & crcHiBit;
        crcPoly_d = (crcPoly_d << 1) & crcMask;
        if (hasHiBit) crcPoly_d ^= crcPoly;
    }
}
//
// Hash function
//
hash_t hash(char const* key)
{
    unsigned char const* k = (unsigned char const*) key;
    hash_t c, currentHash = 0;

    while ( (c = *k++) != 0 )
        currentHash = HashTable[(currentHash>>shiftAmt) ^ c] ^ (
            (currentHash << bytesize) & crcMask);
    return currentHash;
}
```

The ability to give an item a full name that is the same as its original file path name allows files to be moved into and out of a 3SF file without affecting the other items that may reference it. The name reference lookup algorithm ensures that the item will still be found.

Dependency Lists

The load dependency is a list of score items. If this list is present, then the score player software preloads all the items in the load dependency list when it preloads the item itself. Preloading is guaranteed to avoid disk access when an item is used, and this mechanism is designed to support this requirement.

The play-dependency list allows branching between licks. It is a list of other licks, but these licks typically do not produce sound. If the list is present, all licks in this list must be played before the lick itself is played. The purpose of this list is to allow such requirements as program changes and “controller chasing,” which sets all controllers to a certain known state before the lick itself is played.

The dependency list has the syntax:

```
DepsList ::= LoadDeps | PlayDeps
LoadDeps ::= "LDEP" #{ ItemRef* }
PlayDeps  ::= "PDEP" #{ ItemRef* }
```

MIDI Context Format

This section contains details of the MIDI context (MCON) score item. Recall that this item is classified as an axe.

```
Axe ::= "FORM" #{ "AXE " ItemHeader AxeForm }
AxeForm ::= SampleAxeForm | 3INSAxeForm | MIDIContextAxeForm
```

The specific format for an axe that is a MIDI context is

```
MIDIContextAxe ::= "FORM" #{ "AXE " MIDIContextItemHeader
                             MIDIContextAxeForm }
ItemHeader ::= "IHDR" #{ ItemNumber ItemName ItemType [DepsList]
                        ItemKnob* }
MIDIContextItemHeader ::= "IHDR" #{ ItemNumber ItemName 'PIMP' }
MIDIContextAxeForm ::= "FORM" #{ "PIMP" MixerChunk ProgramChunk* }

MixerChunk ::= "MIXR" #{ MaxVoices MaxPrograms MixerChannels }
MixerChannels ::= cstring // string containing number of channels
MaxVoices ::= uint32
MaxPrograms ::= uint32

ProgramChunk ::= "PROG" #{ ProgNum ProgPriority ProgMaxVoices ProgPad1
                          ProgPad2 InstrumentName SampleRef* }
ProgNum ::= uint8
```

```
ProgPriority ::= uint8
ProgMaxVoices ::= uint8
ProgPad1 ::= uint8 // must be zero
ProgPad2 ::= uint32 // must be zero
InstrumentName ::= cstring
```

There is one SampleRef chunk created for each use of a sample by a particular program. There can be many SampleRefs. The chunk specifies a name reference to the sample axe, followed by a list of settings that override the settings within the specified AIFF or AIFC axe form.

```
SampleRef ::= "SMPR" #{SampleName TagArg*}
SampleName ::= NamedRef // reference to AIFF/C sample axe
TagArg ::= { TagID TagData }
TagID ::= uint32
TagData ::= uint32
// These form a Tag list that will be read in by the score
// player and used in a call to CreateSample(). See
// documentation
// for CreateSample() for more details.
```

In summary, a MIDI context axe is processed PIMap. Like any other axe, it has an item header that specifies the item number and name for that axe. MIDI context item headers, however, may not have dependencies or knobs. Following the MIDI context item header is a MIDIContextAxeForm, which encloses all the MIDI context information. This contains a single Mixer chunk, which specifies the type of mixer to be used for the MIDI programs that follow. Each MIDI Program chunk contains a reference to an instrument to be used for that program, and a sequence of Sample References. Each Sample Reference identifies a sample to be used, and all of the settings needed play that sample within the enclosing program context.

MIDI Event Format

In this section, we will give details of the MIDI event (MDEV) score item. Recall that the former is classified as a "Lick."

This section is still under construction.

Index

Numerics

- 2:1 3DO SQXD TSD-56
- 3DO DataStreamer TSD-7
- 3DO instrument TSD-18
- 3DO Score File Format TSD-30
- 3DO Score File Format (3SF) TSD-30
- 3INS TSD-100
- 3SF TSD-32, TSD-40
 - sound design with TSD-32
- 3SF Architecture TSD-30, TSD-35
- 3SF architecture TSD-35
- 3SF file creation
 - from ARIA TSD-32
 - with MakeScore tool TSD-33
- 3SF file format TSD-89, TSD-90
 - all-in-one format TSD-32
- 3SF file playback
 - from ARIA TSD-33
 - from C or C++ program TSD-33
 - from PlayScore TSD-33
- 3SF format TSD-89, TSD-90, TSD-92
- 3SF tool set TSD-40
- 3SF tools TSD-42, TSD-43
 - DumpIFF TSD-42
 - MakeScore TSD-41
 - PlayScore TSD-43
- 4:1 ADPMC compression TSD-3, TSD-56

A

- ADPMC compression TSD-3
- aesthetic considerations TSD-7
- AIFC TSD-100

- AIFC file
 - with SquashSnd TSD-25
- AIFF TSD-76, TSD-99
- AIFF file
 - editing TSD-4
 - from Red Book file TSD-4
 - generating with SoundHack TSD-56
 - preparing TSD-4
 - with SquashSnd TSD-25
- AIFF samples TSD-18
- algorithmic sound effects TSD-6
- Analyze Only option TSD-65
- Apple MIDI Manager TSD-13, TSD-14
- ARIA
 - Comm3DO configuration with TSD-12
 - installing TSD-11
 - playing MIDI file TSD-18
 - required folders TSD-13
 - setup TSD-12
- audio TSD-76
- audio data TSD-7
- audio production TSD-4
- audio *See alsound*
- audio tools function summary TSD-75, TSD-76
- audio/video synchronization TSD-5
- Audiomedia sound file TSD-54
- axes TSD-97

B

- b PIMap flag TSD-19
- Bands option TSD-64
- Bands pop-up TSD-68
- bigfixed TSD-90

Binaural Filter TSD-57, TSD-58

Binaural filter

creating stereo file TSD-57

using TSD-58

C

changing TSD-50

changing current sound file TSD-48

changing header information TSD-50

changing pitch but not file length TSD-64

chunk TSD-92

Close command TSD-55

Comm3DO application

running with ARIA TSD-12

compressed format TSD-48, TSD-49

compressed sound TSD-3

constants TSD-91

Content Library TSD-4, TSD-5

Continue Process command TSD-70

Control menu TSD-70

Continue Process command TSD-70

Pause Process command TSD-70

Show Output command TSD-70

Show Spectrum command TSD-70

Stop Process command TSD-70

convert current sound file to TSD-49

convert current sound file to compressed format
TSD-49

convert sound file to TSD-48

convert sound file to compressed format TSD-48

convert to compressed format TSD-48

converting Apple sound resource to sound file
TSD-56

converting sound file to Apple sound resource
TSD-56

converting sound to AIFF files TSD-56

convolution

multiplying spectra TSD-58

Convolution command TSD-58

Convolution dialog

Filter Gain TSD-59

Impulse window TSD-59

Kilobytes Needed To Process File TSD-59

Length used TSD-59

Moving TSD-60

Ring Modulate TSD-59

create TSD-32

create 3DO score file TSD-36

create score file TSD-32, TSD-33

create score file with TSD-33

creating TSD-6

control functions TSD-65

stereo file TSD-57

creating algorithmic sound effects TSD-6

cstring TSD-91

cstring31 TSD-91

current sound file

changing TSD-48

converting to AIFF format TSD-49

playing TSD-49

D

-d PIMap flag TSD-19

data organization TSD-91

data type TSD-90

data types TSD-90, TSD-91

decompressing sound TSD-3

dependency lists TSD-104

DSP Designer sound file TSD-54

DumpIFF TSD-42

DumpIFF tool TSD-42

E

Edit Function dialog TSD-65, TSD-66

creating control functions TSD-65

default scaling functions TSD-66

Output Waveform dialog TSD-66

Phase Vocoder command TSD-65

Pitch Scale function TSD-65

Time Scale button TSD-65

using TSD-65

Edit menu TSD-56

editing TSD-76

AIFF files TSD-4

editing synthesized TSD-76

editing synthesized audio TSD-76

ErrorProc TSD-44

ErrorProce TSD-44

Export SND Resource command TSD-56

Export SND resource command TSD-56

extended TSD-90

F

-f PIMap flag TSD-19

File menu TSD-54

- Close command TSD-55
- Export SND resource command TSD-56
- Import SND resource command TSD-56
- lisTen to Aiff file command TSD-56
- Open Any command TSD-55
- Open command TSD-54
- Quit command TSD-56
- Save A Copy command TSD-55, TSD-56
- file structure TSD-92, TSD-93, TSD-94, TSD-95, TSD-96, TSD-97, TSD-99, TSD-100, TSD-101, TSD-102, TSD-104, TSD-105
- file strucure TSD-92
- file types and extension TSD-89
- file types and extensions TSD-89
- file version chunk TSD-95
- files in examples folder TSD-76
- Filter window option TSD-64
- flags
 - MIDI PIMap TSD-19
- folder setup TSD-13
- FORM chunk TSD-93
- form chunk TSD-93
- from ARIA TSD-32
- from C++ program TSD-33

G

- Gain Change command TSD-61
- Gain Change dialog TSD-61
- Gain/Reduction option TSD-68
- Gate-Duck pop-up TSD-67
- general MIDI
 - working with TSD-22
- global numeric index TSD-100

H

- h PIMap flag TSD-19
- Hack menu TSD-56
 - Gain Change command TSD-61
 - Header Change command TSD-50
 - Loops & Markers command TSD-51
 - Phase Vocoder command TSD-64
 - Phase Vocoder dialog TSD-65
 - Spectral Dynamics command TSD-66
 - Varispeed command TSD-68
- harmonic relations TSD-7
- Header Change command TSD-50

header file

- modifying markers TSD-51
- header information TSD-50
- headerless file TSD-50, TSD-55
 - saving TSD-55
- Highest/Lowest Band option TSD-68

I

- ID TSD-91
- Import SND Resource command TSD-56
- Impulse Window option TSD-59
- install TSD-48
- installing ARIA TSD-11
- Instruments folder TSD-13
- int16 TSD-90
- int32 TSD-90
- int8 TSD-90
- IRCAM/BICSF sound file TSD-54
- item header TSD-99, TSD-100
- item headers TSD-99
- item names TSD-97
- item numbers TSD-97
- item numbers and item names TSD-97

K

- Kilobytes Needed To Process file option TSD-59

L

- l PIMap flag TSD-19
- Length Used option TSD-59
- licks TSD-97
- lisTen to AIFF file command TSD-56
- listen to AIFF File command TSD-56
- Load PIMap button TSD-21
- load sound file TSD-48
- loading TSD-48
- looping sound effects TSD-5

M

- m PIMap flag TSD-19
- MacMix sound file TSD-54
- MakeScore TSD-33
- MakeScore tool TSD-41
 - using to create 3SF file TSD-33
- mastering TSD-76

- MDEV TSD-99
- MDFL TSD-99
- Microsoft WAVE sound file TSD-54
- MIDI
 - example files TSD-18
 - file TSD-18
 - preparation TSD-18
 - program number TSD-18
 - project file TSD-18
 - project interface tips TSD-23
 - tricks TSD-24
 - working with real-time TSD-23
 - working with real-time MIDI TSD-23
- MIDI context format TSD-104
- MIDI event format TSD-105
- MIDI file TSD-31
 - playing on 3DO station TSD-20
- MIDI File button TSD-21
- MIDI files
 - loading and playing TSD-21
- MIDI folder TSD-13
- MIDI Project window
 - importing PIMap TSD-20
- MIDI scores TSD-6
- modifying
 - header file markers TSD-51
 - loop in sound file TSD-51
 - MIDI file header TSD-51
- multiplying spectra TSD-58
- music
 - creating TSD-5
 - looping TSD-5
- music for 3DO titles TSD-1
- Mutation command TSD-61

N

- name index TSD-102
- New MIDI Project TSD-20
- NeXT sound file TSD-54
- note bounce TSD-14

O

- Open Any command TSD-55
- Open command TSD-54
- opening any file as soundfile TSD-55
- opening headerless and text sound files TSD-55

- opening sound file
 - extensions TSD-54
 - types TSD-54
- OSType TSD-91
- Output Waveform dialog TSD-66
- Overlap option TSD-65
- overview TSD-35

P

- p PIMap flag TSD-19
- PatchBay TSD-13, TSD-14
- PatchDemo TSD-76
- Pause Process command TSD-70
- Phase Vocoder command TSD-64, TSD-65
- Phase Vocoder dialog TSD-64
- Phase Vocoder dialog options
 - Analyze Only TSD-65
 - Bands TSD-64
 - Filter window TSD-64
 - Overlap TSD-65
 - Pitch Scale TSD-65
 - Resynthesis Gating TSD-65
 - Scaling TSD-65
 - Scaling Function TSD-65
 - Time Scale TSD-65
- PIMap
 - parts of TSD-18
 - working with TSD-18
- PIMap file TSD-18
 - example TSD-19
 - flags TSD-19
- PIMP TSD-99
- Pitch Scale Edit Function dialog TSD-66
- Pitch Scale option TSD-65
- play TSD-31
- play 3DO score file TSD-37
- play back score file TSD-33
- play back with PlayScore TSD-33
- Play button TSD-21
- play current sound file TSD-49
- play in DSP TSD-31
- play in memory TSD-31
- play MIDI file TSD-31
- Play, Pause, Stop TSD-45
- playback from ARIA TSD-33
- playing TSD-6
- playing MIDI scores TSD-6
- PlayScore TSD-33, TSD-43

PlayScore tool TSD-43
PlayScore.lib TSD-43, TSD-44, TSD-45
 ErrorProc TSD-44
 ScoreReader TSD-43
preparing TSD-76
preparing AIFF files TSD-4
processing file with TSD-58
project management TSD-8
pstring TSD-91

Q

Quality option TSD-69
Quit command TSD-56

R

RAM-resident vs. ROM-resident sound TSD-2
raw files TSD-49
read/write TSD-49
read/write raw files TSD-49
real-time MIDI TSD-23
 working with TSD-23
Red Book to AIFF TSD-4
Resynthesis Gating option TSD-65
Ring Modulate option TSD-59

S

sample TSD-31
sample rate for sound TSD-3
sample,play in memory TSD-31
sample,spool from disk TSD-31
sample,weave into stream TSD-31
sampled music TSD-5
Samples folder TSD-13
sampling sound effects TSD-4
Save A Copy command TSD-55, TSD-56
saving in 3DO compressed format TSD-56
scaling amplitude of sound file TSD-61
Scaling Function option TSD-65
scaling gain TSD-61
Scaling option TSD-65
score file TSD-32, TSD-33
score file playback TSD-33
score item TSD-101
score items TSD-96
semitones TSD-66
sequencer TSD-14

serial port TSD-14
set header TSD-99
setting header channel number TSD-50
setting header data format TSD-50
setting header sample rate TSD-50
sfStartReading TSD-44
Show Output command TSD-70
Show Spectrum command TSD-70
SMPTE TSD-6
sound
 compressed TSD-3
 decompression TSD-3
 harmonic relations TSD-7
 looping TSD-5
 project management TSD-8
 RAM-resident vs. ROM-resident TSD-2
 sample rate TSD-3
 spooled vs. streamed TSD-2
 stereo vs. mono TSD-3
 synchronization TSD-8
 volume settings TSD-7
sound design TSD-7
sound design with TSD-32
sound effects
 for 3DO titles TSD-1
 sampling TSD-4
sound file TSD-48
 loading TSD-48
 modify loop TSD-51
 playing TSD-48
 scaling amplitude TSD-61
sound set TSD-95, TSD-101
Soundfile Information window TSD-48, TSD-54
SoundHack TSD-48, TSD-54, TSD-56, TSD-70, TSD-75
SoundHack Control menu TSD-70
SoundHack File menu TSD-54, TSD-55, TSD-56
SoundHack Hack menu TSD-57, TSD-58, TSD-61, TSD-64, TSD-66, TSD-68
specifying TSD-101
specifying score item TSD-101
specifying sound set TSD-101
Spectral Dynamics command TSD-66
Spectral Dynamics Processor dialog TSD-67
 Bands pop-up TSD-68
 Gain/Reduction TSD-68
 Gate-Duck pop-up TSD-67
 Highest/Lowest Band TSD-68
 Threshold Level TSD-68

Spectral Mutation dialog TSD-61
spool from disk TSD-31
Spooled vs. streamed sound TSD-2
SquashSnd
 caveats TSD-27
 compatible file types TSD-25
 compression/decompression parameters
 TSD-26
 convert raw file TSD-25
 MPW tool TSD-25
 parameters TSD-26
squashsnd TSD-3
SquashSound TSD-75
standard dynamics processing
 compression TSD-66
 ducking TSD-66
 expansion TSD-66
 gating TSD-66
 Spectral Dynamics command TSD-66
StartReading TSD-44
StartReadingAndPlaying TSD-44
stereo vs. mono TSD-3
Stop Process command TSD-70
Stop Processing command TSD-70
streaming TSD-7
streaming audio data TSD-7
streaming audio data with TSD-7
Sun sound file TSD-54
synchronization TSD-6
synchronizing audio and video TSD-5
syntax definitions TSD-94
synthesized audio TSD-76
synthetic patch TSD-31

T

testing TSD-6
Threshold Level option TSD-68
Time Scale option TSD-65

U

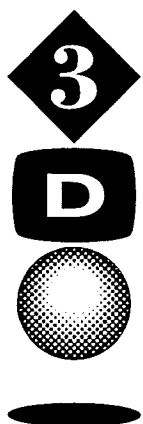
uint16 TSD-90
uint32 TSD-90
uint7 TSD-90
uint8 TSD-90
using TSD-40
using binaural filter TSD-58

V

Varispeed box TSD-69
Varispeed command TSD-68
Varispeed dialog TSD-69
 Quality TSD-69
 Varispeed Function TSD-69
 Varspeed box TSD-69
 Vary by Pitch/Scale TSD-69
Varispeed Edit Function dialog TSD-68, TSD-69
Varispeed Function option TSD-69
versionstamp TSD-91
volume settings TSD-7

W

weave into stream TSD-31
with MakeScore TSD-33
working with TSD-5
working with real-time MIDI TSD-23



3DO M2 Audio and Music Programmer's Guide

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Audio and Music Programmer's Guide

Preface

About this Book.....	MPG-xvii
About the Audience	MPG-xvii
How this Book Is Organized	MPG-xvii
Related Documentation	MPG-xviii
About the Code Examples.....	MPG-xix
Typographical Conventions	MPG-xix

1

How To Write Audio Software

How Do I Make a Simple Sound?	MPG-2
How Do I Play an Audio Sample?	MPG-2
How Do I Determine When a Sample or Envelope Finishes Playing on a Specific Instrument?	MPG-3
How Do I Play a Long Sound File?	MPG-3
How Do I Play a Musical Score?.....	MPG-4
How Do I Manage Multiple Sound Effects?	MPG-5
How Do I Create Complex Sound Effects?.....	MPG-6
How Do I Make a Sound Fade Out Smoothly without Popping?	MPG-6

2

Understanding 3DO Audio

What is 3DO Audio?	MPG-8
Audio Hardware.....	MPG-8
The CD Player	MPG-9
Audio Software.....	MPG-10
Designing Sound with the Audio Folio	MPG-10

Playing Notes	MPG-13
Using the Music Library	MPG-15

3

Playing a Synthetic Sound

Introduction	MPG-19
The Process of Using Instruments	MPG-20
Accessing the Audio Folio	MPG-21
Instruments and Templates	MPG-21
Template	MPG-21
Instrument	MPG-22
Kinds of Instrument Templates	MPG-22
Standard Instrument Templates	MPG-22
Instrument Ports	MPG-25
Preparing Instruments	MPG-26
Loading a Standard Instrument Template	MPG-26
Creating a Mixer Template	MPG-27
Creating an Instrument	MPG-28
Calculation Rates	MPG-29
Instrument Resources	MPG-30
Loading a Standard Instrument	MPG-30
Connecting Instruments	MPG-31
Connecting One Instrument to Another	MPG-31
Disconnecting One Instrument From Another	MPG-33
Attachments	MPG-33
Playing Instruments	MPG-34
Starting an Instrument	MPG-34
Releasing an Instrument	MPG-37
Stopping an Instrument	MPG-38
Cleaning Up When Finished	MPG-38
Introducing the Audio Clock	MPG-40
Reading the Global Audio Clock	MPG-40

4

Playing a Sampled Sound

Introduction	MPG-43
Sampled Sound	MPG-44
Loading and Attaching Samples	MPG-46
Loading Samples from Disk	MPG-46

Sample Loops	MPG-50
Sample Trigger Points	MPG-51
Attaching a Sample to an Instrument.....	MPG-51
The Attachment Item	MPG-53
Attaching Multisamples to an Instrument	MPG-53
Detaching a Sample from an Instrument	MPG-54
Deleting a Sample	MPG-54
Debugging a Sample	MPG-54
Example Program	MPG-55
Modifying Attachments.....	MPG-55
Setting an Attachment's Attributes	MPG-55
Specifying a Start Point	MPG-55
Setting a Start-Independent Attachment	MPG-56
Creating an Instrument-Stopping Attachment	MPG-56
Independently Controlling Attachments.....	MPG-57
Starting an Attachment	MPG-57
Releasing an Attachment	MPG-57
Stopping an Attachment	MPG-57
Linking Attachments.....	MPG-58

5 Controlling Sound Parameters

Introduction.....	MPG-61
Digital Signal Processing Signal Types.....	MPG-61
Audio and Control Signals	MPG-61
Signal Flow Between Connected Instruments	MPG-62
Control Signal Arithmetic	MPG-63
Knobs and Probes	MPG-64
Type Casting for Knobs and Probes	MPG-66
Handling Knobs.....	MPG-67
Finding Knobs	MPG-67
Creating a Knob	MPG-69
Finding Knob Parameters	MPG-70
Setting Knob Values	MPG-70
Reading Knob Values	MPG-71
Controlling Knobs with Generic Values	MPG-71
Deleting a Knob	MPG-73
Creating and Attaching Envelopes	MPG-74
Envelope Properties	MPG-74

Creating an Envelope	MPG-83
Attaching an Envelope to an Instrument	MPG-83
Detaching an Envelope from an Instrument	MPG-85
Deleting an Envelope	MPG-85

6

Advanced Audio Folio Usage

Introduction	MPG-87
Reading and Changing Audio Item Attributes	MPG-87
Reading Audio Item Characteristics	MPG-88
Setting Audio Item Characteristics	MPG-88
Reading Instrument Output	MPG-89
Creating the Probe	MPG-89
Using Probes	MPG-89
Tuning Instruments	MPG-90
Creating a Tuning	MPG-90
Applying a Tuning	MPG-92
Deleting a Tuning	MPG-92
Bending Pitch	MPG-93
Adding Reverberation	MPG-94
A Delay Line Overview	MPG-94
Creating a Delay Line	MPG-95
Deleting a Delay Line	MPG-96
Connecting a Delay Line to Create Reverberation	MPG-96
Using a Delay Line for Oscilloscope Data	MPG-99
Audio Clocks	MPG-99
Creating a Custom Audio Clock	MPG-99
Setting the Clock Rate	MPG-100
Function Calls	MPG-100
Tuning	MPG-100
Adding Reverberation	MPG-101
Audio Clocks	MPG-101

7

Patch Templates

Introduction	MPG-103
Patch Compiler	MPG-104
Binary Patch File Loader	MPG-104
Makepatch	MPG-104

Patch Examples	MPG-104
Reverberation Examples	MPG-110

8

3D Sound Spatialization

Introduction	MPG-117
Chapter Overview	MPG-117
Sound Cues	MPG-117
Amplitude Panning	MPG-118
Delay Panning	MPG-118
Filtering	MPG-118
Doppler	MPG-118
Distance Factor	MPG-119
Directionality	MPG-119
Locating 3D Sound Examples	MPG-120
Using 3D Sound	MPG-120
Setting up 3D Sounds	MPG-121
Pseudo Code example of 3D Sound	MPG-122
Starting 3D Sounds	MPG-122
Moving 3D Sounds	MPG-123
Deleting 3D Sounds	MPG-124

9

Sound Player

An Overview of the Sound Player.....	MPG-125
Simple Sound Player Example	MPG-126
Players, Sounds, and Markers	MPG-128
Players	MPG-128
Sounds	MPG-128
Markers	MPG-128
Examples of Looping and Branching	MPG-129
Decision Functions	MPG-130
Decision Functions and Actions	MPG-132
Example Decision Function	MPG-132
Rules for Decision Functions	MPG-133
Caveats	MPG-134
For More Information	MPG-134
Function Calls	MPG-134
Player Management Function Calls	MPG-134

Sound Management Function Calls	MPG-135
Marker Management Function Calls	MPG-135
Decision Function Calls	MPG-136
Debug Function Calls	MPG-137
Constants	MPG-137

10

Using the Sound Spooler

How the Sound Spooler Works.....	MPG-140
How to Use the Sound Spooler.....	MPG-140
An Example.....	MPG-141
Convenience Functions	MPG-145
ssplSpoolData()	MPG-145
ssplPlayData()	MPG-146
Sound Spooler Function Calls.....	MPG-146
Convenience calls	MPG-147
SoundSpooler management calls	MPG-147
SoundBufferNode management calls	MPG-147
Low level calls	MPG-148
Callback routine	MPG-148

11

Playing MIDI Scores

A Brief MIDI Review.....	MPG-151
MIDI Channels	MPG-151
Channel Messages	MPG-152
MIDI Score Playback	MPG-152
Creating an Internal MIDI Environment.....	MPG-153
Creating a Virtual MIDI Device	MPG-153
Importing a MIDI Score	MPG-155
Providing MIDI Playback Functions	MPG-156
Setting the Audio Clock Speed to Control Tempo	MPG-157
An Overview of the MIDI Score-Playing Process	MPG-157
Setting Up.....	MPG-158
Creating a MIDI Environment.....	MPG-158
Setting Voice and Program Limits	MPG-158
Creating a Score Context	MPG-159
Setting PIMap Entries	MPG-160
Setting Up a Mixer	MPG-164

Importing a MIDI Score	MPG-165
Declaring a MIDIFileParser Data Structure	MPG-165
Creating a Juggler Object	MPG-165
Loading the Score	MPG-165
Specifying the User Context	MPG-166
The Interpreter Procedure	MPG-167
Setting the Tempo	MPG-168
Setting the Audio Clock Rate	MPG-168
Playing the MIDI Score	MPG-168
Dynamic Voice Allocation	MPG-170
Freeing Instruments Created by Voice Allocation	MPG-171
Using MIDI Functions	MPG-172
Interpreting and Executing a MIDI Message	MPG-172
Starting and Releasing a Note	MPG-173
Directly Starting and Releasing an Instrument	MPG-174
Changing a Program	MPG-175
Setting Channel Panning and Volume	MPG-175
Bending Pitch	MPG-176
Changing Characteristics During Playback	MPG-178
Cleaning Up	MPG-179
Creating a MIDI Score for Playback	MPG-180
Primary Data Structures	MPG-181
Function Calls	MPG-181
MIDI Environment Calls	MPG-181
MIDI Score Calls	MPG-182
MIDI Playback Calls	MPG-182

12

Creating and Playing Juggler Objects

Object-Oriented Programming Concepts	MPG-184
Objects	MPG-184
Methods and Messages	MPG-184
Object Variables	MPG-185
Classes and Instances	MPG-185
Creating New Classes	MPG-186
Managing Juggler Objects	MPG-186
Initializing the Juggler	MPG-186
Defining a New Class	MPG-187
Creating an Object	MPG-187

Destroying an Object	MPG-187
Checking an Object's Validity	MPG-188
Default Juggler Classes	MPG-188
The Jugglee Class	MPG-188
The Sequence Class	MPG-189
The Collection Class	MPG-190
Creating Sequences and Collections	MPG-190
Creating a Sequence	MPG-191
Creating a Collection	MPG-195
Sending Messages to Sequences and Collections	MPG-198
Calling Methods Directly	MPG-198
Message Macros	MPG-198
Messages for Sequences	MPG-198
Messages for Collections	MPG-201
Playing Sequences and Collections	MPG-203
A Juggler Operational Overview	MPG-203
The Juggler Process	MPG-205
Bumping the Juggler	MPG-205
Terminating the Juggler	MPG-206
An Example Program	MPG-206
Function Calls and Method Macros	MPG-210
Juggler Control Calls	MPG-210
Object Management Calls	MPG-211
Method Macros	MPG-211
Object-Defining Tag Arguments	MPG-211

13

Tips and Techniques

DSP Resources	MPG-214
Dynamic Voice Allocation	MPG-214
Audio Samples	MPG-214
Available RAM for Sampling	MPG-214
Looping Stereo and Mono Sound	MPG-214
Translating PC VOX files	MPG-214
Loading Multiple Sound Files	MPG-215
Playing Scores	MPG-215
Streaming Audio versus Score Playback	MPG-215
Score Files	MPG-216
MIDI	MPG-217

Playing Red Book Audio	MPG-217
Timing	MPG-217
DSP Time versus System Time	MPG-217
Multi-thread Timers	MPG-218
Filters.....	MPG-218
Modulating Selected Frequency Ranges	MPG-218
Miscellaneous.....	MPG-218
Allocating Signals	MPG-218

14

Beep Folio

Differences Between Beep Folio and Audio Folio	MPG-219
Reasons for Choosing Audio Folio	MPG-221
Reasons for Choosing Beep Folio	MPG-221
What Beep Folio Doesn't Provide	MPG-221
Beep Folio Vocabulary	MPG-222
Machine	MPG-222
Voice	MPG-222
Channel	MPG-222
Parameter	MPG-222
Steps in Using the Beep Folio	MPG-222
Other Considerations	MPG-223
Mixing	MPG-223
Cache Coherency	MPG-223
Knowing when a sample is finished playing	MPG-223

List of Figures

Figure 2-1 Audio input and output processing.	MPG-8
Figure 5-1 Amplitude modulation.	MPG-64
Figure 5-2 Frequency modulation.	MPG-66
Figure 5-3 Measuring a repeating waveform	MPG-72
Figure 5-4 An example envelope shape	MPG-75
Figure 5-5 Envelope loops.	MPG-78
Figure 6-1 Sample of a delayed audio signal.	MPG-95
Figure 6-2 A typical reverberation setup using a delay line and a delay instrument.	MPG-97
Figure 7-1 1tap.mp	MPG-111
Figure 7-2 multitap.mp	MPG-112
Figure 7-3 lowpass_comb.mp	MPG-113
Figure 7-4 multi_lowpass.mp	MPG-114
Figure 7-5 allpass_filter.mp	MPG-115
Figure 7-6 allpass_filter.mp Control Path Subpatch	MPG-115
Figure 8-1 Rustling Tree and Trumpet for 3DO Directionality Examples.	MPG-119
Figure 8-2 Simplest case of a stereo mixer for 3D sound	MPG-120
Figure 8-3 3D Sound Space.	MPG-121
Figure 8-4 The polar axis for PolarPosition4D	MPG-123
Figure 9-1 Playing sounds in a continuous sequence.	MPG-126
Figure 13-1 Beep Machine.	MPG-220

List of Tables

Table 3-1	Some of the <code>CreateInstrument()</code> Tag arguments	MPG-28
Table 3-2	Some of the <code>StartInstrument()</code> Tag arguments that set frequency.	MPG-36
Table 3-3	Some of the <code>StartInstrument()</code> Tag arguments that set amplitude.	MPG-36
Table 4-2	Tag arguments that define the characteristics of a sampled sound.	MPG-49
Table 4-3	Tag arguments that define the characteristics of an attachment.....	MPG-52
Table 5-1	<code>InstrumentPortInfo</code> members.....	MPG-68
Table 5-2	<code>CreateKnob()</code> tag arguments.....	MPG-70
Table 5-3	Knob query tag arguments.....	MPG-70
Table 5-4	<code>EnvelopeSegment</code> members.....	MPG-75
Table 5-5	Tags associated with the Envelope's <code>EnvelopeSegment</code> array.....	MPG-75
Table 5-6	Tags to control Envelope sustain and release loops.....	MPG-76
Table 5-7	Tags to control Envelope time scaling.....	MPG-80
Table 5-8	Other Envelope Tags.....	MPG-82
Table 5-9	Envelope flags.....	MPG-82
Table 5-10	<code>CreateAttachment()</code> tag arguments for envelopes.....	MPG-83
Table 5-11	Attachment flags for Envelopes.....	MPG-84
Table 12-1	Employee object variable values.....	MPG-185
Table 12-2	Example event list.....	MPG-189

Preface

About this Book

The *3DO Audio and Music Programmer's Guide* describes the Audio and Music folios. It includes overviews, programming tutorials, sample code, and function call definitions. The overview chapters help you understand the concepts behind the function calls; the programming chapters describe the procedures to use with those calls.

About the Audience

This book is written for application developers. To use this document, you should have a working knowledge of the C programming language, object-oriented concepts, and, of course, music.

How this Book Is Organized

This book contains the following chapters:

Chapter 1, How To Write Audio Software, provides a starting point and road map from which you can begin writing audio software.

Chapter 2, Understanding 3DO Audio, introduces you to the hardware and software behind the 3DO system's audio features.

Chapter 3, Playing a Synthetic Sound, shows you how prepare and play an instrument.

Chapter 4, Playing a Sampled Sound, shows you how to load and attach sound samples to an instrument.

Chapter 5, Controlling Sound Parameters, describes how to modify sound characteristics by using knobs and envelopes to modify frequency or amplitude.

Chapter 6, Advanced Audio Folio Usage, describes how to change the characteristics of audio items, and tune instruments.

Chapter 7, Patch Templates, describes how to construct custom patch templates for use with the Audio Folio and how to create complex sound effects by combining instruments together in a patch.

Chapter 8, 3D Sound Spatialization, describes ways to make a sound appear to be coming from a point in space and to move from one place to another.

Chapter 9, Sound Player, shows you how spool large sound files from disc.

Chapter 10, Using the Sound Spooler, describes how to use the low level system sound spooler to spool blocks of sound data from memory.

Chapter 11, Playing MIDI Scores, discusses techniques that use the Music library to import a MIDI score from a standard MIDI file and then play the score back using audio folio resources.

Chapter 12, Creating and Playing Juggler Objects, describes the object-oriented programming environment of the Juggler, and how to use the Juggler to schedule events, such as a musical score, in time.

Chapter 13, Tips and Techniques, provides a question-and-answer section of common problems encountered when using the Music library and Audio folio.

Chapter 14, Beep Folio, introduces you to the low overhead alternative to the Audio Folio.

Related Documentation

The following additional manuals are useful to developers who are programming in the 3DO environment.

3DO System Programmer's Guide is a guide to the features of the kernel, I/O, file system, and so on in the 3DO operating system.

3DO System Programmer's Reference contains a detailed description of the calls that make up the Portfolio system.

3DO Graphics Programmer's Guide describes the graphics engine that makes up the Graphics folio. It includes overview chapters, programming tutorials, sample code, and graphic function call definitions.

3DO Graphics Programmer's Reference contains a detailed description of the calls that make up the Portfolio Graphics folio.

3DO Debugger Programmer's Guide is a guide to using the 3DO Debugger. It provides a tutorial on using the debugger as well as descriptions of the graphical user interface.

3DO DataStreamer Programmer's Guide is a guide to the 3DO DataStreamer architecture, streaming tools, and weaver tool.

3DO DataStreamer Programmer's Reference provides manual pages for the structures in the 3DO DataStreamer library, for all data preparation tools, and for all weaver script commands.

3DO Tools for Sound Design provides background on sound design for a 3DO system and discusses Patchdemo, ARIA, and SoundHack.

About the Code Examples

The examples used in this book are provided in electronic format in the Examples folder of the released CD. See the README file in the Examples:Music folder for a complete listing of all example programs.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	Err OpenAudioFolio(void)
function name	CreateAudioClock()
new term or emphasis	That added weight is called a <i>cornerweight</i> .

How To Write Audio Software

This chapter is the starting point for writing audio software and addresses basic audio programming tasks. It is provided first, to give you an outline of the necessary steps from which to learn. It does not tell you everything you need to know, but instead points you to the chapter that contains the complete description of a topic. When you see a function mentioned, look up that function in Chapters 1 and 2 of the *3DO Audio and Music Programmer's Reference* for complete information.

This chapter contains information on the following topics:

Topic	Page Number
How Do I Make a Simple Sound?	2
How Do I Play an Audio Sample?	2
How Do I Determine When a Sample or Envelope Finishes Playing on a Specific Instrument?	3
How Do I Play a Long Sound File?	3
How Do I Play a Musical Score?	4
How Do I Manage Multiple Sound Effects?	5
How Do I Create Complex Sound Effects?	6
How Do I Make a Sound Fade Out Smoothly without Popping?	6

How Do I Make a Simple Sound?

To produce a sound, do the following:

1. Load one or more digital signal processor (DSP) instruments by calling `LoadInstrument()`.

DSP instruments are programs that run on the DSP. They provide the functionality of oscillators, filters, envelopes, sample playback, and so on.

2. Connect the inputs and outputs of the instruments by calling `ConnectInstruments()`.

This action is similar to patching a modular synthesizer.

3. Connect to an instrument that can make the sound audible, such as `line_out.dsp`.

4. Call `StartInstrument()` to start executing the DSP instrument.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for descriptions of the calls.

See `tone.c` in the *Examples/Audio/Misc* folder for a complete example of making a simple sound.

See Chapter 5, "Controlling Sound Parameters," for more information about envelopes.

How Do I Play an Audio Sample?

To play an audio sample, do the following:

1. Call `LoadSample()` to load an AIFF sample into memory.

You can also load samples of other formats into memory yourself and turn them into 3DO sample items by calling `CreateSample()`.

2. Call `LoadInstrument()` to load a DSP instrument that can play that sample. You can determine the appropriate instrument name by looking in the table at the beginning of Chapter 3, "Instrument Templates," in the *3DO Audio and Music Programmer's Reference*, or by calling `SampleItemToInsName()`.

3. Call `CreateAttachment()` to attach the sample to the instrument.

You can use the same sample on multiple instruments, or you can use multiple samples on one instrument.

4. Connect to an instrument to make the sound audible, such as

`line_out.dsp.`

5. Call `StartInstrument()` to start playing the sample.

The instrument starts whatever sample is attached to it.

6. Call `ReleaseInstrument()` to stop a sample that is looping,

For More Information

For more information on the calls in this section, see Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference*.

For an example of playing a sound, see the example program *playsample.c*.

How Do I Determine When a Sample or Envelope Finishes Playing on a Specific Instrument?

You can connect samples and envelopes to a specific instrument by creating an attachment item. Use the same sample on multiple instruments, or use multiple samples on one instrument. What you really need to know is when a specific attachment has finished. There are two ways to do this:

- ◆ The first is to call `MonitorAttachment()` to request a signal when the sample or envelope has finished. This method can be used with `WaitSignal()` which lets a task sleep until the attachment is finished.
- ◆ The second way is to set the `AF_ATT_FATLADYSINGS` flag in the attachment. When the attachment finishes, it stops the associated instrument. You can then poll the instrument by calling `GetAudioItemInfo()` with `AF_TAG_STATUS`.

Be careful when you are polling for an instrument to finish. Never loop continuously while polling (this is called *busy waiting*). Busy waiting takes up CPU cycles and can cause a drop in performance or a deadlock.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for information on these calls.

See the example files *ta_attach.c* and *simple_envelope.c* for examples of `MonitorAttachment()` usage.

How Do I Play a Long Sound File?

If you have a long AIFF sound file that does not fit in memory, you need to play it as it is read from disc. There are several ways to do this:

- ◆ Use the advanced sound player. It lets you play multiple sound files, as well as samples in memory, branches between files, and loops seamlessly without

glitches. See the `spCreatePlayer()` function call and Chapter 7 "Advanced Sound Player," for more information. For examples, see *tsp *.c*.

- ◆ If you need to read other information from disc, along with the audio, you can use the `DataStream`. It allows you to mix various types of data like audio, video, application data, and so on. The various data types are arranged in "chunks" of like data, that are then read from the disc and passed to the corresponding "subscribers." Subscribers are tools that process a particular type of data.
- ◆ If you need to build your own sound file player from scratch, you may want to use the low-level sound spooler in the Music library. You can get the audio data into memory any way you want, then pass it to the sound spooler to be queued and ultimately played using the DSP. See *ta_spool.c* for an example.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for information on these calls.

See the *DataStream Programmer's Guide* and *DataStream Reference Guide* for more information on the `DataStream`.

How Do I Play a Musical Score?

Playing a musical score can be done as follows:

1. Use commercial sequencers to compose the score, and then export it as a Format 0 or Format 1 MIDI file.
2. Create a PIMap text file that maps MIDI program numbers to AIFF sample files or ARIA instrument files.

The PIMap text file is required so that the score player knows what to do when it encounters a MIDI program change command.

3. Load the score into Juggler Objects called Sequences and Collections. See *Examples/Juggler/*.c* for Juggler examples.

Sequences are arrays of events. In this case, the events are MIDI commands with time stamps. The *Collections* hold multiple Sequences played in parallel.

4. Once the score is loaded, you can start playing it by calling `StartObject()`.

There are a number of things you can do with a score while it is playing: You can mute tracks, change the tempo, or pause. You can also request that callback functions be called when the score starts, repeats or stops. You can use the callback functions to chain together multiple score fragments for interactive score playing.

For More Information

See Chapter 11, “Playing MIDI Scores,” for programming details.

For an example of playing a musical score, see the example program *playmf.c*.

How Do I Manage Multiple Sound Effects?

There are several ways to manage multiple sound effects, depending on the nature of the effects.

If you have a few Samples, and they can be played by instruments that fit in the DSP at the same time, you can do the following:

1. Load all of the instruments and samples.
2. Attach the samples.
3. Connect all of the instruments to a mixer.
4. Call `StartInstrument()` to trigger the sound effect for the appropriate instrument.

If the instruments do *not* all fit in the DSP at the same time, but the Samples are all of the same format, you can do the following:

1. Load several instruments and connect them to a mixer.
2. Load all of the samples.
3. Attach all of the samples to each instrument using `CreateAttachment()` with the following flags set: `AF_ATTFF_NOAUTOSTART|AF_ATTFF_FATLADYSINGS` using `SetAudioItemInfo()`, so that you can start the instrument without triggering any particular sample.
4. Keep track of the attachments returned by `CreateAttachment()` in arrays.
5. Start each of the instruments playing.
6. Call `StartAttachment()` when you want to trigger a sample on a particular instrument for the appropriate attachment.

Call `GetAudioItemInfo()` with `AF_TAG_STATUS` to determine which instrument is stopped when you want to trigger the next attachment. See “How Do I Play a Long Sound File?” later in this chapter for more details.

If you have multiple kinds of instruments, you need a more sophisticated dynamic voice allocation scheme. You can use the voice allocator that is part of the score-playing utility in the Music library. See *Examples/SoundEffects/sfx_score.c* for an example of a dynamic voice allocation scheme.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for descriptions of the calls.

How Do I Create Complex Sound Effects?

To create complex sound effects, load instruments separately by calling `LoadInstrument()` and connect them together using `ConnectInstruments()`. Then use `SetKnob()` to control the sound parameters. This method lets you create complex patches out of any existing instrument.

- ◆ The second option is to use `MakePatch` which lets you create custom DSP instrument templates out of a library of existing instruments. These custom instruments can be loaded with a call to `LoadPatchTemplate()`. See the contents of `Examples/Audio/Patches` for several examples of `MakePatch` patches.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for descriptions of the calls.

How Do I Make a Sound Fade Out Smoothly without Popping?

Pops are caused by a sudden change in amplitude. A pop can occur if you call `StopInstrument()` when an instrument is still sounding, or if you set an amplitude or gain knob abruptly to 0.

To avoid pops, you need to smoothly change the amplitude from full amplitude to 0, over at least 10 msec, to eliminate the pop. This can be done in two ways:

- ◆ You can call `SetKnob()` rapidly in a loop. This ties up the main CPU but does not require any special DSP resources.
- ◆ You can use an instrument that generates a slowly-changing value such as `envelope.dsp`, `integrator.dsp`, or `slew_rate_limiter.dsp`.

For More Information

See Chapter 1, "Audio Folio Calls," in the *3DO Audio and Music Programmer's Reference* for descriptions of the calls.

See Chapter 3, "Instrument Templates," in the *3DO Audio and Music Programmer's Reference* for descriptions of the instrument templates.

See the example file `ta_nopop.c`. This example program can be obtained from the 3DO InfoServer.

Understanding 3DO Audio

This chapter introduces you to the hardware and software behind the 3DO system's audio features. We start with an overview of the audio hardware to help you understand how the Audio folio creates sound. The chapter also provides an overview of the Audio folio, introducing the concepts behind sound creation and score playback.

This chapter contains the following topics:

Topic	Page Number
What is 3DO Audio?	8
Audio Hardware	8
Audio Software	10

What is 3DO Audio?

3DO audio is a collection of audio features centered around two pieces of hardware: the CD player, that provides standard Red Book CD audio and sampled sound files, and the digital signal processor (DSP), that manipulates sounds from the CD player and synthesizes sounds of its own.

The 3DO Portfolio controls the audio hardware with:

- ◆ The Audio folio – a collection of function calls that provide low-level audio control. Low-level audio calls control the DSP to synthesize or play back sound data, or to modify the way the sound is synthesized or played back.
- ◆ The Music library – a collection of high-level music-related calls. High-level music calls assemble and play back hierarchical musical scores, and play them back in synchronization with animation or in response to player input.

Audio Hardware

Like most audio systems, the 3DO system receives incoming audio signals and transmits outgoing audio. Between input and output, it processes the incoming audio or generates new audio. Figure 2-1 shows standard 3DO audio input, output, and processing.

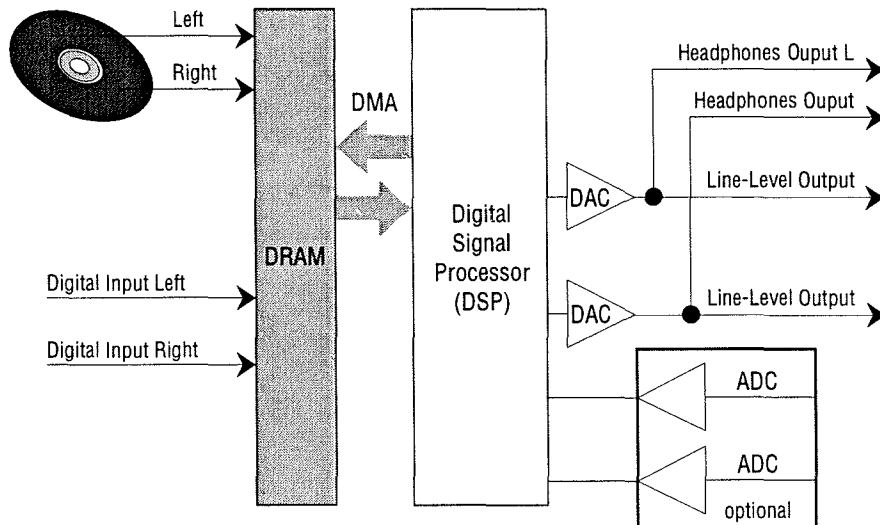


Figure 2-1 Audio input and output processing.

The 3DO system can accept audio input from the following sources:

- ◆ A built-in compact disc (CD) player, that reads the standard Red Book audio CD format, as well as most standard CD data formats.
- ◆ Direct digital input, that brings audio signals into the DSP for the purpose of processing and mixing.
- ◆ Any digitized audio placed in DRAM, such as SlipStream.

The 3DO system can send audio out through these features:

- ◆ Two line-level output jacks connected to DACs (Digital-to-Analog Converters), that convert 16-bit digital sampled sound into analog signals.
- ◆ A player-bus stereo headphone jack, which accepts the analog output of the DACs and provides a headphone-level signal on a 3DO controller. This allows you to hear 3DO audio through connected headphones.
- ◆ Any external device that can read DRAM and take digitized audio stored there.

To process audio, the 3DO system uses:

- ◆ The digital signal processor (DSP), a programmable processor built into the BDA chip. The DSP provides 32 bidirectional DMA channels of digital input, performs standard math operations on the input data, and provides two channels of digital-to-analog audio output.

To provide additional audio control, the 3DO system can read and send MIDI messages, if the system contains the following optional expansion equipment:

- ◆ MIDI In, Out, and Thru ports connected to the 3DO system expansion port.

The CD Player

The 3DO system's internal CD player is designed to play back standard Red Book audio CDs with full 16-bit precision and bandwidth at a sample rate of 44,100 samples per second. When a Red Book CD is played, its samples pass directly through the DSP. The DSP does not process those samples, but simply fetches left and right samples and passes them to the appropriate left or right DAC so that they put out stereo audio to an amplified speaker system or to headphones.

The fundamental code necessary to play back a Red Book CD is stored in the 3DO system ROM. You can simply turn on a 3DO system, place a CD in the drive, press a play button, and listen to the CD. Simple Red Book playback does *not* require a CD containing the Portfolio boot files.

The CD player also reads Yellow Book CDs, spinning the CD at double the standard rate to provide faster data access. Portfolio can read sampled sound files from these CDs; the files are typically stored using the AIFF standard maintained by Electronic Arts and Apple Computer. These sampled-sound files can be fed directly to the DSP.

Audio Software

The 3DO system depends on three sources of software to control its audio hardware:

- ◆ The Red Book playback code in ROM, which is used only for playing Red Book CDs without Portfolio boot files.
- ◆ The Audio folio, which generates and processes sound. It plays back sound samples, synthesizes new sounds, and can modify both sampled and synthesized sounds.
- ◆ The Music library, which imports and creates scores, and plays back large sampled sound files. Scores are ordered assemblies of sounds and other components that a task can play back later. A score can typically be music, collections of sound effects, drawing or animation functions, or combinations of different elements. A score can also be an imported MIDI score.

Designing Sound with the Audio Folio

Working with sound is a fairly abstract process. You cannot see the sound, and many of the modifications you make to sound in the digital domain do not have simple parallels in the real world. For example, think of changing the reverb time of an instrument. In the strictly acoustic world, the only parallel is moving to another room or, even less intuitive, rebuilding the room around you. To make these operations more intuitive, the Audio folio bases its sound generation and processing on the earliest and easiest to understand synthesizer: the modular analog synthesizer.

A modular synthesizer is a collection of modules and patch cords. Each module can produce an electric signal or modify an incoming electric signal. For example, one module produces a sawtooth wave signal. Another module accepts the sawtooth signal and then filters out all high frequencies. Yet another module accepts the filtered sawtooth signal along with three other incoming signals, merges them, and sends out a stereo pair of signals.

Analog synthesizer modules typically have input ports to accept signals, output ports to send signals, and controls (quite often knobs) that control the actions of the module. Combining the work of the modules creates rich signals and sends them out to the speaker system, where they're translated into sounds. The patch cables combine the modules. For example, the output of the sawtooth wave generator is connected by cable (it is "patched") to the low-pass filter, which

filters out high frequencies. The resulting signal is patched into a ring modulator module that adds some grit to the signal, and then passes it on (via another cable) to a mixer that adds the signal to other signals. The mixer then passes the results out (via more cables) to the amplified speaker system. The output sound is the result of the combination of modules patched together by cables.

As a sound passes through a patched set of modules, you can change the settings of the individual modules – modifying the quality of the sound. For example, the module combination described above may include a knob on the low-pass filter module. When the knob is turned up or down, the filter cutoff point moves up or down, letting more or less high frequencies through. As you turn that knob up and down, the resultant sound gets more or less shrill as higher frequencies are passed or filtered out.

The Audio folio provides the equivalent of modules, controls, and cables with *instruments*, *knobs*, and *attachments*. Each instrument generates or modifies a digital signal; each knob on an instrument controls the way the instrument works; and each attachment connects (patches) the output of one instrument with the input or knob of another instrument. Audio folio calls work like the hands of a musician, making attachments between instruments and finally sending the output to the DACs, where it's converted to an analog signal and sent on to amplified speakers. Creative instrument connections and control make rich and variable audio possible.

Instruments

Each instrument used by the Audio folio is created externally and provided in a large library of instruments that comes with the 3DO Toolkit™. Those instruments can be combined to create new instruments using software development tools such as ARIA. The new instruments can then be used alongside the standard instruments.

An instrument is stored on a CD (or on the Macintosh™ in a development system) and, when needed, it is loaded into DRAM. The Audio folio allocates DSP resources to handle the instrument and makes the instrument available for tasks such as playing or modifying sounds, or to connect to other instruments.

Instruments can vary from simple to complex. Some are simply mixers or reverb units; others are completely self-contained sound synthesis environments, comparable to a commercial digital synthesizer. Each instrument can contain the following types of elements:

- ◆ Inputs, which accept a signal from another instrument.
- ◆ Outputs, which send a signal from the instrument to another instrument, or to the final audio output.
- ◆ Knobs, which accept control input for the instrument.

- ◆ Sampled-sound pointers, which point to a sampled sound waveform table that the instrument can play.

The number of inputs, outputs, knobs, and sampled sound pointers available to an instrument is limited only by the resources available in the DSP, but a successful instrument should be frugal. An instrument with too many inputs, outputs, knobs, and sampled-sound pointers is a resource hog and cannot coexist with other instruments on the DSP.

A task *must* use an instrument to play any kind of sound or note. For example, to play back a sampled sound, a task must use a sampled sound instrument. This is a very simple instrument that points to the sampled sound and plays the sound whenever instructed. The task can then send the sampled sound through other instruments for modification, or it can send its output directly to the DACs. When a task uses a more complex instrument, such as a full-blown synthesizer, it often doesn't need to use additional instruments to fatten the sound; the synthesizer has enough internal capabilities to come up with the necessary sound.

Each instrument, no matter how simple or complex, is DSP code that runs during each DSP cycle to create an ordered set of sample values that represent sound. When the Audio folio allocates DSP resources to an instrument, it assigns an input DMA channel for each sampled sound input, an internal (I) memory location for each standard input and output, an output DMA channel for optional digital output, and an external input (EI) memory location for each instrument knob. Memory is also allocated for programs and DSP "ticks" or instructions cycles. These resources can be exhausted, thereby limiting how many instruments you can play at the same time. When multiple instruments play, their code is located in instruction (N) memory, daisy-chained together in order of priority, and their other resources are allocated appropriately.

Knobs

Each instrument knob controls an instrument parameter. Knob parameters are defined when the instrument is defined by the audio workshop tool. Some knobs control simple parameters, such as frequency or amplitude. Others control more abstract parameters, such as the timbre used in synthesis. Each knob is identified by a name.

When an instrument is first loaded into the 3DO system, all knob-controlled parameters are set to default values. To change any of those parameters, a task must first *grab* the knob for the parameter it wants, by specifying the knob's name in the audio call `CreateKnob()`. The task then *tweaks* the knob to the setting it wants. Knob tweaking can be a one-time event to create a new and steady setting, or it can be a continuous event to create a changing setting. For example, a task can tweak a filter knob up once to create more shrill sounds, or it can move the filter knob up and down to create a sound that audibly alternates between shrill and mellow.

If, at any time, a task needs to check on the setting of a knob, it can *peek* at the knob and get a return value of the knob setting. When the task is finished with the knob, it releases the knob, which retains its setting value at the time of its release.

Digital Audio Samples

Some instruments can play digital audio samples from RAM, using one of the DMA/FIFO channels. Samples are played by attaching them to instruments. A sample can be loaded from an AIFF file on disc, created from scratch, or can come from any other possible 3DO source. Samples can also be included with the instrument definition. Note that a sampled-sound instrument *cannot* play sampled sound files or tables directly from a compact disc.

Many instruments contain multiple sampled sound pointers, so that they can use a different sampled sound for different pitch ranges. For example, a sampled piano instrument can use one sampled sound for its lowest octave, another sampled sound for the next octave up, and so forth for each octave of its range. Multiple samples avoid stretching or shrinking the original sample too much for higher and lower pitches. This makes playback more realistic.

Playing Notes

Once an instrument is loaded and connected, it does nothing until instructed to play a *note*. The word “note” implies a pitched sound of a certain duration, part of a musical score, but that is not the case in the Audio folio. A note here can be any sound, from the plucking of a guitar string to the rumble of a race car. Notes are used to play music and also to provide sound effects. For example, a sampled sound instrument can be set to play a “boom” whenever an explosion appears on screen. When a task asks the instrument to play a note, the instrument sends a “boom” out through the speakers.

Note Stages

The quality of a note over time is set in the instrument definition. However it is shaped, a note must have three stages: a start, a release, and a stop. For example, consider a trumpet note. When the note starts, there’s a “puh” sound followed by the blare of the trumpet. That blare holds until the note releases. On release, the blare quickly diminishes until the note stops and there is no more sound.

As another example, consider a sampled sound playback. When the note starts, playback starts and can loop over through a section of the waveform table. When the note releases, the waveform table plays through to the end, where the note stops.

In the Audio folio, start, release, and stop commands to an instrument mean the following:

- ◆ *Start*—asks an instrument to begin note execution, which can mean beginning sampled-sound playback, new sound generation, or any other event meant to occur during the note.
- ◆ *Release* —asks an instrument to finish note execution, which can mean playing a sampled sound table to the end, tapering off a synthesized sound, or immediately stopping the sound. What happens on release depends on how the instrument is designed to play the note.
- ◆ *Stop* —asks an instrument to stop note execution completely and immediately, so that the instrument no longer produces a sound. The difference between stop and release is that a stopped note stops sounding immediately and ignores any of the rest of the note. A released note can continue playing after the release request until the note is finished.

The concept of start, release, and stop stages correspond to the stages of a key press on a piano or similar keyboard. When a key goes down, the note starts—the string vibrates, a pipe blows, or an oscillator starts. When the key comes up, the note is asked to finish—the string is damped, the pipe stops blowing, or the oscillator stops. Because most instruments do not finish vibrating immediately, the vibrations die out and the note stops some time after the key release.

To play a note on an instrument, a task uses the audio calls `StartInstrument()`, `ReleaseInstrument()`, and `StopInstrument()` to tell an instrument to start, release, or stop a note. The instrument's response to each of the three commands depends on how the note is defined and shaped within the instrument. Some instruments can have a predefined note definition (such as a sampled-sound player), others can have knobs that let a task change the note definition (something like envelope controls on a standard synthesizer).

As a note progresses after its start or release, any time-based process, such as an envelope, is timed by audio ticks. These ticks are generated 240 times per second by the DSP, and can be sped up to a maximum of around 1000 times per second.

Specifying Pitch and Loudness

When a task signals an instrument to start a note, it can optionally specify a frequency and amplitude for the note, which directly sets its pitch and loudness. It can also optionally specify pitch or velocity as given in MIDI scores. The velocity information determines amplitude, envelope shape, or other sound qualities that are set by the instrument's definition. These qualities are usually those affected when you strike a music keyboard key harder or softer.

The pitch information is fed first to a *tuning table*, which contains a frequency for each of the standard half-step pitches in all audible octaves. The indexed frequency goes to the instrument, which then uses the frequency to play the note.

The tuning table, by default, offers 12-tone equal-tempered tuning, but allows many other tuning systems as well for music playback. If a task wants to avoid standard half-step tuning, it can send frequencies directly to the instrument, bypassing the tuning table.

Voices

The ability to play a single note at one time is called a *voice*. Each instrument has a single voice. To play more than one note at a time (for musical harmony or multilayered sound effects), a task must use more than one instrument at a time. For example, you can use multiple instances of the same instrument to create a chorus of like-timbred voices.

To play notes simultaneously, a task simply asks for a new note to start on one instrument before other instruments have stopped playing their notes.

Using the Music Library

A task can use Audio folio calls directly, to time and play notes on instruments, but it is simpler to use higher-level Music library calls to take care of these jobs. Music library calls fall into these categories:

- ◆ *Sound spooler* lets you continuously send audio data to the Audio folio for playback.
- ◆ *Sound player* plays large sampled sound files from disc without having to load the entire sample file into memory at one time. (It uses the sound spooler).
- ◆ *Juggler calls* create and assemble *scores* (which are collections of events), and then play those scores back using a scheduling mechanism called the Juggler.
- ◆ *MIDI file-parser calls* read a format 0 or format 1 MIDI score file from disc, and translate the contents into a score playable by the Juggler.
- ◆ *MIDI interpreter calls* create a virtual MIDI environment and then translate standard MIDI messages into Audio folio calls that produce the proper MIDI-specified results. These calls are commonly used as a simple sound manager to simplify playing sound effects.

Spooling Sound Files

Audio folio functions allow a task to play back sampled sounds that are stored in RAM. Unfortunately, being RAM-resident limits the length of the sampled sound: a very long sample can easily use all available RAM and still need more. The Music library's sound-spooling functions allow a long sample to be stored on disc, safely out of RAM, and then spooled into RAM, a small section at a time, for continuous and uninterrupted playback.

To spool sounds, the Music library borrows an idea from double-buffered animation: it reads from one section of RAM while it writes to a different section of RAM. To do so, it creates several sound buffers in RAM, each capable of storing a section of the sample file, each capable of playback through a sampled-sound instrument. These sound buffers are linked together and are played in circular order: from the first through to the last, then repeated from first to last over and over, as long as requested.

While circular playback of the sound buffers occurs, other calls spool a disc-based sound file a section at a time into unplayed buffers, just as double-buffered animation draws into undisplayed graphics buffers. Writing spooled sample sections into sound buffers in circular order keeps a step ahead of reading from those buffers. The result is smooth playback of a large sample file on disc. Because the spooling process runs at high priority, it is very unlikely that playback will be interrupted by the activities of other tasks.

Playing Scores

At the heart of the Music library is a family of calls that support the Juggler, a scheduling mechanism that handles lists of timed events and executes them at the proper time. To work properly, the Music library supports an object-oriented programming environment, where a task can create Juggler objects containing scores (collections of events) or other Juggler objects. These objects can be assembled into hierarchical structures and respond to method calls changing their properties.

Without getting into details (which are described in Chapter 10 "Creating and Playing Juggler Objects"), the Juggler and its programming environment support the following features:

- ◆ Scores that can include musical events (such as notes), or nonmusical events (such as calls to draw figures on the screen), retrieve information from CD, and so on.
- ◆ Objects that are a collection of subobjects. This allows one object to be a collection of scores that can be played back sequentially or in parallel.
- ◆ Real-time object manipulation so that a score or score collection can change structure in response to a task's requests. A score or collection changed during playback changes the quality of playback. This is useful if a task wants to change the musical ambience during the course of its execution.
- ◆ The ability to define new object classes (including their methods) so that a task can work with custom Juggler objects.

Note: *This feature is not available in the current release of Portfolio.*

- ◆ Completely abstract event timing so that a score can be played back using the strict time available from the audio timer (or similar time source), or using a task's own generated time, which can depend on changing conditions within the task.

Playing MIDI Score Files

Musical scores for computers are typically created using MIDI sequencing software, available on development computer platforms. These scores can be saved as Format 0 or Format 1 MIDI files, and then stored on CD-ROM for playback using Music library functions. The Music library's MIDI-handling functions can import a Format 0 or Format 1 MIDI score from disc, translate it into a Juggler object, and then store it in RAM.

To play the MIDI commands, stored in a MIDI Juggler object, other Music library calls can create a virtual MIDI environment in software. That MIDI environment uses data structures to specify audio instruments assigned to respond to different MIDI messages, to keep track of notes played and voices used, to keep track of overall volume and stereo mix, and to handle other parameters commonly changed during playback of a MIDI score.

Once a MIDI score is translated and stored as a Juggler object, and a MIDI environment is created to handle playback, the Juggler can execute the MIDI events stored in those objects. Each MIDI event contains a standard MIDI message, which a MIDI translation function translates. When a MIDI translation function translates a MIDI message, it makes the appropriate Audio folio calls to carry out the message. It also keeps track of its activity in the MIDI environment's data structure. For example, when the Juggler reads a MIDI *Note On* message in a MIDI event, it calls a *Note On* translation function that first finds the appropriate audio instrument to play the note, and then starts the instrument and records the fact that it is playing in the appropriate data structure.

See Chapter 11, "Playing MIDI Scores," for more detailed information about MIDI score playback.

Playing a Synthetic Sound

This chapter gives you the basic steps you need to play a synthetic sound using the audio folio. It contains the following topics:

Topic	Page Number
Introduction	19
The Process of Using Instruments	20
Accessing the Audio Folio	21
Instruments and Templates	21
Connecting Instruments	31
Playing Instruments	34
Introducing the Audio Clock	40

Introduction

When you program with the Audio folio family, you work with instruments. Portfolio instruments play notes—sound events that range from a traditional musical note with pitch and duration, to a sound effect such as an explosion, to a straightforward playback of a sampled sound file. Portfolio instruments also work on other instruments, mixing them together, adding special effects such as filtering and reverberation, and controlling their playback.

The model for handling Portfolio instruments is the modular analog synthesizer. You can add instruments (the equivalent of synthesizer modules) to the mix, connect instruments together for combined effects (the equivalent of connecting modules), you can adjust each instrument's controls to change playback

characteristics over time, and you can trigger each instrument to start, release, and stop notes. The Audio folio's job is to control the full synthesizer, setting up music and sound effects.

The Process of Using Instruments

To use instruments, a task must first open the Audio folio then load and prepare the instruments. After the instrument environment is set, the task plays notes on the instruments. When it is finished, the task deletes all instruments and audio items it created and closes the Audio folio. The overall process is as follows:

1. Open the Audio folio.
2. Prepare the instruments by loading an instrument template from disc.
 - ◆ Create an instrument from the instrument template, allocating RAM and audio resources.
 - ◆ Set the tuning of any instrument or template (optional).
 - ◆ Load a sample and attach it to the instrument (if it's a sampled sound instrument).
 - ◆ Create and attach an envelope to an instrument (optional).
 - ◆ Connect the output of one instrument to the input of another instrument (optional).
 - ◆ Connect the audio outputs of all instruments to one or more mixers (mandatory if you want to hear the instruments' output).
3. Grab any of an instrument's knobs for later adjustment (optional).
4. Set knobs to set an instrument's starting characteristics (optional).
5. Play the instruments.
 - ◆ Start a note on an instrument.
 - ◆ Wait a specified period of time.
 - ◆ Set knobs to change an instrument's operation while playing (optional).
 - ◆ Release a previously-started note (sometimes optional).
 - ◆ Stop a previously-started note (sometimes optional).
6. Independently play an instrument's attachment, such as an envelope (optional) by starting, releasing, and stopping the attachment.
7. Clean up when finished:
 - ◆ Delete all no longer needed instruments.
 - ◆ Delete any no longer needed instrument templates.
 - ◆ Delete all other no longer needed Audio folio items.
 - ◆ Close the Audio folio.

As you see, there are several optional steps. For example, many tasks do not have any need to set an instrument's tuning; they use the 12-tone, equal-tempered default tuning.

You should also note that the order shown is not fixed, especially when it comes to playing an instrument. For example, a task can set instrument knobs before, during, or after note playing on the instrument. A task can also start a note on one instrument, then load and allocate a new instrument.

Nonetheless, there is *some* fixed order to the process: a task must load an instrument template before it can create an instrument; it must create an instrument before it can connect the instrument, grab and tweak knobs, or play notes on the instrument; and it must load and connect a sample to a sampled sound instrument before playing the instrument. Beyond the fact that audio items must be created before they are used, most Audio folio order is set by common sense and taste.

See *tone.c* and *playsample.c* for examples of using instruments.

Accessing the Audio Folio

The Audio folio is demand-loaded. To make Audio folio calls, a task must first open the Audio folio with this call:

```
Err OpenAudioFolio (void)
```

The function returns a non-negative value if successful. If unsuccessful, it returns a negative number (an error code). If you do not open the Audio folio before you use audio calls, your task will abort and an error message will be displayed.

Instruments and Templates

The modular synthesis approach used by the audio folio permits you to produce simple or complex sounds by connecting together blocks called *Instruments*. Like other Portfolio objects, Instruments are represented by Items. Instrument Items are created from Template Items. The distinction between Templates and Instruments is like the difference between an object-oriented programming class and an object.

Template

A Template is an Item which defines the qualities of an Instrument—how many inputs, knobs, and outputs it has; what signals it generates; how it passes input signals to outputs; and so on. Each template is, in fact, a DSP program. Templates are merely definitions, and do not actually run on the DSP. That is the job of Instruments.

Instrument

Instruments represent executing instances of the DSP program defined by a template. They may be started and stopped independently, and may have various characteristics (such as frequency and amplitude) which can be adjusted dynamically.

The reason for having this two-step hierarchy is that it facilitates creating multiple voices from a single sound definition by encapsulating the sound definition into a Template from which you may create an arbitrary number of Instruments. For example, this technique can be used to build a multi-voice synthesizer, where the voice description is contained in the Template, and each voice consists of an Instrument created from that Template. Each Instrument representing a voice might play at an different pitch, amplitude, or any other runtime characteristic defined by the Template.

Kinds of Instrument Templates

There are three kinds of Instrument Templates:

- ◆ **Standard.** Portfolio comes with a set of predefined instrument templates that are stored in the system directory on disc. A list of the various categories of standard templates is given in Standard Instrument Templates. All of the standard templates are described individually in Chapter 3, "Instrument Templates," in the *3DO Music and Audio Programmer's Reference*.
- ◆ **Mixers.** Mixers are instruments that mix audio signals from other instruments in a manner similar to an audio mixing board. The result can be connected to further processing or connected to the line out. Because of the enormous range of possible configurations, there isn't any way to satisfy everyone with a set of statically defined mixer instruments. So mixers are constructed on demand based on a simple set of parameters, primarily number of inputs and number of outputs.
- ◆ **Patches.** The audiopatch folio permits constructing custom instrument templates out of existing instrument templates. Such patches may be created within a program or loaded from a patch file written by tools such as `makepatch`. Patches are described in Chapter 7.

Standard Instrument Templates

The standard Portfolio instruments fall into several categories:

- ◆ **Sampled sound instruments** play sampled sounds by reading from a DMA channel. Samples and sample player instruments are covered in more detail in Chapter 4, "Playing a Sampled Sound."
- ◆ **Sound synthesis instruments** synthesize audio signals from scratch.

- ◆ **Effects instruments** typically accept audio signals from other instruments and alter those signals in ways that alter the original signal (such as filtering, distortion, delay).
- ◆ **Control signal instruments** generate a low-frequency signal typically used to adjust the knobs of other instruments. Examples of these include envelopes and low frequency oscillators (LFOS).
- ◆ **Arithmetic instruments** perform simple mathematical operations on signals. Examples of these include add, subtract, multiply, minimum and maximum.
- ◆ **Line in and out instruments** interface between DSP signals and the audio line in and out jacks.

The standard instrument templates are referred to by name, all of which end in `.dsp`.

Sampled Sound Instruments. The largest number of standard instruments are sampled sound instruments, which can use a variety of techniques to play back sampled sound tables stored in the AIFC or AIFF formats (which can be created by most commercial sound editing programs). See Table 4-1 for more information on these instruments.

Sound Synthesis Instruments. Portfolio's standard sound synthesis instruments generate their own audio signals instead of reading them from sample data. Some of those instruments are:

- ◆ `triangle.dsp` generates a triangle-wave signal.
- ◆ `sawtooth.dsp` generates a sawtooth-wave signal (and has a grittier sound than a triangle-wave signal).
- ◆ `noise.dsp` generates a white-noise signal.
- ◆ `rednoise.dsp` generates a more gritty noise signal than `noise.dsp` generates.
- ◆ `impulse.dsp` generates impulse waveforms.
- ◆ `pulse.dsp` generates pulse waveforms.
- ◆ `square.dsp` generates square waveforms.

Effects Instruments. Effects instruments typically accept an audio signal, alter it, and pass the altered signal out. In the case of delay-effects instruments, they accept an audio signal and pass it out through DMA to memory, where it can be altered by another instrument. Portfolio's effects instruments include:

- ◆ `svfilter.dsp` accepts a signal, filters it, and sends the result to its output. `svfilter.dsp` is a state-variable filter, which has knobs that control frequency, resonance, and amplitude. It has low-pass, band-pass, and high-pass outputs.

- ◆ `cubic_amplifier.dsp` amplifies an incoming signal using a cubic function that results in distortion similar to a guitar fuzz box.
- ◆ `delay_f1.dsp` and `delay_f2.dsp` accept a signal and writes the signal directly to a sample buffer, where it can be reread to create a reverb loop. (This is discussed in a later chapter.)

Control Signal Instruments. Control signal instruments generate a control signal, typically a low-frequency signal too slow to be heard by itself but useful for controlling instrument knobs. Sound synthesis instruments can be used as control signal instruments if they are set to a very low frequency and then connected to the knob of another instrument. For example, a triangle wave generator can be set to a low frequency and then connected to the frequency knob of a filtered noise generator to produce a wind sound that rises and falls with the frequency of the triangle wave generator.

The standard control-signal instruments include:

- ◆ `envelope.dsp` can be used to generate multi-segment envelopes or as a ramp generator. This is commonly used to control such time-variant parameters as amplitude, filter cutoff frequency, etc.
- ◆ `pulse_lfo.dsp` uses extended precision arithmetic to give lower frequencies than `pulse.dsp`. It also has better resolution at the same frequency. The frequency range of this instrument is 256 times lower than its corresponding high-frequency version. It is useful as a modulation source for controlling other instruments, or for bass instruments.
- ◆ `envfollower.dsp` tracks the positive peaks of an input signal. It outputs a fairly smooth signal that can be used to control other signals.
- ◆ `randomhold.dsp` generates new random numbers at a given rate and holds steady until a new number is chosen.
- ◆ `triangle_lfo.dsp` is a triangle wave generator that uses extended precision arithmetic to give lower frequencies than `triangle.dsp`. It also has better resolution at the same frequency.
- ◆ `square_lfo.dsp` is a square wave generator that uses extended precision arithmetic to give lower frequencies than `square.dsp`. It also has better resolution at the same frequency.

Arithmetic Instruments. These instruments perform simple arithmetic operations on signals. Some of the standard arithmetic instruments are:

- ◆ `add.dsp` and `subtract.dsp` perform clipped, signed addition and subtraction between their two inputs.
- ◆ `multiply.dsp` and `multiply_unsigned.dsp` are general-purpose signed and unsigned multipliers. They each output the product of their two inputs.

In addition to the obvious, `multiply.dsp` can also be used for signal processing operations such as signal attenuation and ring modulation.

- ◆ `timesplus.dsp` performs a single-operation multiply and accumulate on its three inputs: $A * B + C$. This is useful for LFO modulation.
- ◆ `minimum.dsp` accepts two inputs and outputs the smaller of the two. `maximum.dsp` accepts two inputs and outputs the larger of the two. These instrument can be used for clipping.
- ◆ `schmidt_trigger.dsp` permits signalling the host CPU when a signal crosses a threshold.

Line In and Out Instruments. These instruments interface with the audio line in and out jacks.

- ◆ `line_out.dsp` sends a stereo DSP signal to be accumulated with other output instruments and ultimately to the audio line out jacks.

Note: *If the accumulated output signal ever exceeds the maximum of 1.0, it is clipped back to 1.0. This can result in horrible distortion, so it is important to keep the final results down to an acceptable level.*

- ◆ `line_in.dsp` outputs the stereo Audio Input signal from the optional ADC.

Note: *Each task that intends to use `line_in.dsp` instrument must successfully enable audio input with `EnableAudioInput()` before the instrument can be created.*

- ◆ `tapoutput.dsp` permits reading the accumulated stereo output from all currently running output instruments.

Instrument Ports

All Instruments have a set of named *ports* (inputs, outputs, and knobs) for connecting with other instruments and controlling parameters. Inputs and outputs are for connecting instruments to one another. Knobs are for adjusting parameters. Each port has one or more *part*. Parts are used to represent multiple channels of an port and to functionally group audio signals. For example, an instrument which outputs a monophonic signal has an output port with one part, an instrument which outputs a stereo signal has an output port with two parts, a 6 x 2 mixer has an input port with six parts and an output port with two parts. Parts are specified as integers in the range of 0 to `numParts-1`. Part numbers for stereo ports may be specified using the handy definitions `AF_PART_LEFT` and `AF_PART_RIGHT`, which are defined as 0 and 1 respectively.

All ports have an associated *signal type*, which defines the floating-point units of the port. Most ports are operate in the range of -1.0 to 1.0, but some are calibrated in units which are more convenient for the function of the port. For example, frequency knobs are calibrated in Hertz.

The port set for each of the standard instrument templates is presented in Chapter 3, "Instrument Templates," of the *3DO Audio and Music Programmer's Reference*. Also, the 3DO shell program `insinfo` lists all of the ports of any instrument.

Preparing Instruments

Each of the three kinds of templates has its own creation method:

- ◆ **Standard.** Standard instrument templates are loaded using `LoadInsTemplate()`.
- ◆ **Mixers.** Mixer templates are created on demand with `CreateMixerTemplate()`.
- ◆ **Patches.** `CreatePatchTemplate()` and `LoadPatchTemplate()` are ways to create patch templates. They are described in the *3DO Audio and Music Programmer's Reference*. Designing Sound with Patch Templates is described in Chapter 7, "Patch Templates" of this book.

There is just one way to create an Instrument from any kind of Template:
`CreateInstrument()`.

Loading a Standard Instrument Template

Standard Instrument Templates are stored on disc in the `System.m2` directory. The following function is used to load a standard instrument template:

```
Item LoadInsTemplate (const char *insName, const TagArg *tagList)
```

where `insName` points to a string containing the name of the instrument. Since the standard instrument templates are in a known location within the system directory, `insName` should only contain a simple file name, not a full path name. For example,

```
insTemplate = LoadInsTemplate ("sawtooth.dsp", NULL);
```

The function returns the item number of the template if successful, or if unsuccessful, returns a negative number (an error code).

The first time that a template is loaded, the corresponding `.dsp` file is read from the disc. As long as the template remains loaded, all subsequent calls to that type of `.dsp` file type use the version in memory like a cache. Preloading the template lets you avoid disc accesses during time-critical parts of your application. This behavior is system-wide, not merely for each task.

Creating a Mixer Template

Mixers are constructed on demand based on a simple set of parameters:

- ◆ Number of inputs.
- ◆ Number of outputs.
- ◆ Whether there should be a master amplitude knob.
- ◆ Whether the mixed output should be available at a port called “Output,” or sent to the audio line out accumulator (in the same manner as `line_out.dsp`).

These characteristics are encoded into a 32-bit descriptor called a `MixerSpec` using the macro `MakeMixerSpec()`:

```
MixerSpec MakeMixerSpec (uint8 numInputs, uint8 numOutputs,
                        uint16 flags)
```

Once you have a `MixerSpec`, create the Mixer template using `CreateMixerTemplate()`:

```
Item CreateMixerTemplate (MixerSpec mixerSpec,
                        const TagArg *tags)
```

For example,

```
{
    MixerSpec mixerSpec;
    Item mixerTmp;

    mixerSpec = MakeMixerSpec (kNumInputs, kNumOutputs,
                              AF_F_MIXER_WITH_LINE_OUT);
    mixerTmp = CreateMixerTemplate (mixerSpec, NULL);
    .
    .
    .
}
```

For another example, see *Examples/Audio/Misc/ta_customdelay.c* on the 3DO Portfolio M2 Release 2.0 CD.

Once you have the Mixer Template, you may create Instruments from it, embed it in a patch, etc. The `MixerSpec` is also used for other mixer-related operations, notably accessing the Gain knob, so it's a good idea to keep it around.

Each mixer has an Input port with a part for each requested input channel, an optional output port with a part for each requested output channel, and a multi-part Gain knob to adjust the amount of each input part to send to each output part.

The parts of the Gain knob may be thought of as the elements of a two-dimensional matrix, where one dimension selects the input part and the other selects the output part. In order to pick the right knob part, use the macro `CalcMixerGainPart()`:

```
int32 CalcMixerGainPart (MixerSpec mixerSpec, int32 inChannel,
                        int32 outChannel)
```

Creating an Instrument

Once an instrument template has been created as an item, a task can use the template to create an instrument defined by the template. To do so, use this call:

```
Item CreateInstrument (Item InsTemplate, const TagArg *tagList)
```

The call accepts the item number of a previously created instrument template. The function also accepts optional tag arguments, some of which are listed in Table 3-1. These are discussed in greater detail below.

Table 3-1 *Some of the `CreateInstrument()` Tag arguments*

Tag Name	Description
AF_TAG_PRIORITY	Priority of new instrument in range of 0 to 200. Defaults to 100.
AF_TAG_CALCULATE_DIVIDE	Specifies the denominator of the fraction of the total DSP cycles on which this instrument is to run. The valid settings for this are: 1 - Full rate execution (44,100 cycles/sec) 2 - Half rate (22,050 cycles/sec) 8 - 1/8 rate (5,512.5 cycles/sec) Defaults to 1.

When executed, `CreateInstrument()` creates an instrument item defined by the instrument template and allocates the resources the instrument requires. The call returns the item number of the new instrument if successful; if unsuccessful, it returns a negative value (an error code).

Note that you can create as many instruments as you like from a single instrument template. In fact, some tasks can be set up to create new instruments whenever the task does not have enough voices to play the desired notes. For

example, if a task needs to play a four-voice chord but has created only three instruments of the same kind to play that chord, it can create one more instrument to play the chord.

Instrument priority determines the system-wide order of execution of instruments. This is an integer in the range of 0 (the lowest priority) to 200 (the highest priority). Higher priority instruments execute before lower priority instruments. Instruments of the same priority have no guaranteed execution order. Normally instrument execution order isn't significant, but there are times when it is (e.g., feedback loops, digital filters, etc.). For most cases, you should use the default priority.

Calculation Rates

Each DSP instrument corresponds to a DSP program that normally executes once per sample frame. This means that DSP instruments can update their output values at a rate of 44100 Hz, allowing for high fidelity output. However, we often use instruments that generate slowly changing signals, like LFOs or envelopes. These slow instruments do not need to be updated at a rate of 44100 Hz. We can optionally execute these instruments at a lower rate, meaning that they consume fewer DSP cycles. Running instruments at a lower rate can help you fit more instruments into the DSP.

If you have sample data that was recorded at 22050 Hz, you can play it back using a fixed-rate sampler, such as `sampler_16_f1.dsp`, running at half rate. Since the instrument only executes at a rate of 22050 Hz, the sample will be played back at its proper rate. Otherwise, you would have to use a `sampler_16_v1.dsp` with a detune of 0.5, which is more expensive.

Output instruments, such as `line_out.dsp`, may also be run at half rate if necessary. To improve the fidelity of half-rate output instruments, the audio folio performs linear interpolation to convert their accumulated results to the 44100 Hz sample rate of the DAC.

Note: *This interpolation is only performed for output instruments. There is no interpolation done for connections between half-rate and full-rate instruments.*

You can specify the execution rate with the `AF_TAG_CALCULATE_DIVIDE` tag when you create the instrument. A tag value of 2 selects 1/2 rate. A value of 8 selects 1/8th rate. A value of 1 for full rate is the default. Any other values are illegal.

Instrument Resources

The audio folio allocates DSP resources as instruments are created from templates and frees resources as instruments are deleted. If a resource runs out, the audio folio prevents further instrument creation until some instruments are deleted. These resources include DSP code and data memory, hardware resources (e.g., DMA channels), and DSP time usage measured in ticks.

DSP ticks are DSP time units used during each frame of the DSP output. A DSP frame is the time the DSP takes to generate one stereo pair of samples to be sent to the DAC FIFO. The DAC outputs samples at 44,100 samples per second (approximately once every 22.7 microseconds). Because the DAC has an eight-sample FIFO, the DSP can take longer than 22.7 microseconds to produce one sample pair, as long as it takes less than 181 microseconds to produce eight sample pairs. For this reason, ticks are allocated from a pool of batch ticks (i.e., the total number of ticks in an eight-frame batch). Currently, the DSP has approximately 11,200 ticks per batch (about 1400 ticks per frame) available for application use.

Instrument tick consumption is documented in ticks per frame; the system-wide availability of ticks is reported in ticks per batch. The `AF_TAG_CALCULATE_DIVIDE` tag controls the number of frames per batch in which each instrument executes: full rate executes in all eight frames, half rate executes in four of the eight frames, 1/8th rate executes in one of the eight frames. To compute the batch tick consumption of an instrument, multiply the documented ticks per frame by the number of frames per batch in which the instrument runs.

The resources required by each of the standard instrument templates is presented in Chapter 3, "Instrument Templates," of the *3DO Audio and Music Programmer's Reference*. The 3DO shell program `insinfo` lists resource information for any instrument. The 3DO shell program `audioavail` displays the currently available DSP resource amounts. It may be run at any time to see the instantaneous resource load of all created instruments.

Loading a Standard Instrument

To combine the process of loading a standard instrument template and the process of creating an instrument from that template in a single call, you can use this convenience call:

```
Item LoadInstrument (const char *insName, uint8 calcRateDivider,
                    uint8 priority)
```

where `insName` points to a string containing the name of the instrument. The second parameter, `calcRateDivider`, uses the following values:

- ◆ 0, 1 - full rate

- ◆ 2 - half rate
- ◆ 8 - 1/8th rate

All other values are illegal. Apart from supporting 0, this is the same as `AF_TAG_CALCRATE_DIVIDE`. The third parameter, `priority`, is the value for the `AF_TAG_PRIORITY` tag. For most circumstances, use the default priority of 100.

For example,

```
instrument = LoadInstrument ~("sawtooth.dsp", 0, 100);
```

which is essentially equivalent to:

```
insTemplate = LoadInsTemplate ("sawtooth.dsp", NULL);
instrument = CreateInstrumentVA (insTemplate,
    AF_TAG_CALCRATE_DIVIDE, 1,
    AF_TAG_PRIORITY,      100,
    TAG_END);
```

When `LoadInstrument()` executes, it loads the specified instrument template, and creates an instrument from that template. It returns the item number of the instrument if successful, or a negative number (an error code) if unsuccessful.

The instrument template loaded with this call remains in memory, but there is no item number to use to unload it from memory. To do so, you must use the `UnloadInstrument()` call (see the *3DO M2 Audio and Music Programmer's Reference*).

Connecting Instruments

After you have created and set instruments, you can connect them together for combined effects, such as applying an envelope from `envelope.dsp` to the amplitude of another instrument. You can connect instruments together to mix sounds, such as combining a sawtooth instrument and a sampled-sound instrument in a mixer. And you can connect instruments to an output instrument (e.g., `line_out.dsp`) so that their audio signals are fed to the DAC, a necessary step if the instrument is to be heard at all.

Connecting One Instrument to Another

When you connect one instrument to another, you can connect one output of the first instrument to either an input or a knob of the second instrument. When an output of one instrument is connected to an input of a second, the second instrument usually acts *upon* the incoming signal from the first instrument. For example, if you connect the output of `sawtooth.dsp` to the input of

`svfilter.dsp`, `svfilter.dsp` filters the signal generated by `sawtooth.dsp`. The filtered signal is available at `svfilter.dsp`'s output. This output may in turn be connected to another instrument input, and so on.

When an output of one instrument is connected to a knob of a second, the second instrument usually acts *according to* the incoming signal from the first instrument. For example, the output of a slowly oscillating triangle wave generator (`triangle_lfo.dsp`) connected to the frequency knob of `sawtooth.dsp` causes the `sawtooth.dsp` generator to create a sawtooth wave that moves up and down in pitch (i.e., vibrato).

Remember that all instrument ports have one or more parts (e.g., the two channels of a stereo input or output). Connections are made between parts using the function:

```
Err ConnectInstrumentParts (  
    Item srcInstrument, const char *srcPortName, int32 srcPartNum,  
    Item dstInstrument, const char *dstPortName, int32 dstPartNum)
```

The source part is specified by the first three arguments: `srcInstrument` is the item number of the source instrument (the first instrument), `srcPortName` points to a string containing the name of the source instrument's output used for the connection, and `srcPartNum` is the part number of the output port of the source instrument. The last three arguments specify the destination input or knob part to connect to: `dstInstrument` is the item number of the destination instrument (the second instrument), `dstPortName` points to a string containing the name of the destination instrument's input or knob used for the connection, and `dstPartNum` is the part number of the input port of the source instrument to connect to. For example,

```
/* Connect the output of a sawtooth oscillator to the stereo  
** input of line_out.dsp  
*/  
ConnectInstrumentParts (  
    osc_sawtooth, "Output", 0,  
    lineout, "Input", AF_PART_LEFT);  
ConnectInstrumentParts (  
    osc_sawtooth, "Output", 0,  
    lineout, "Input", AF_PART_RIGHT);
```

For connections between single-part ports, you can pass zero in the `srcPartNum` and `dstPartNum` arguments, or use the convenience macro `ConnectInstruments()`, which does just that.

When `ConnectInstrumentParts()` executes, it connects the specified output part of the source instrument to the specified input or knob part of the destination instrument. If successful, it returns a non-negative value. If unsuccessful, it returns a negative value (an error code).

Note that if you connect to a knob part of the destination instrument, that knob part may no longer be adjusted (i.e., `SetKnobPart()` or `StartInstrument()` tag arguments applied to that knob part have no effect) until the connection is broken.

A single output part can be connected to multiple knob or input parts (referred to as *multiple fan-out*). However, a single input or knob part can only be connected to a single output (referred to as *single fan-in*).

Disconnecting One Instrument From Another

To disconnect a source instrument's output from a destination instrument's input or knob, use this call:

```
Err DisconnectInstrumentParts (
    Item dstInstrument, const char *dstPortName, int32 dstPartNum)
```

This call accepts three arguments. Note that they are identical to the three "dst" arguments used with the `ConnectInstrumentParts()` call. The first, `dstInstrument`, is the item number of the destination instrument (the second instrument); the second, `dstPortName`, points to a string containing the name of the destination instrument's input or knob used for the connection; and the third, `dstPartNum`, is the part number of the input port of the source instrument to connect to. For example,

```
/* Break the connections made in the previous example */
DisconnectInstrumentParts (lineout, "Input", AF_PART_LEFT);
DisconnectInstrumentParts (lineout, "Input", AF_PART_RIGHT);
```

When it executes, `DisconnectInstrumentParts()` breaks the specified connection. It returns a non-negative value if successful, and a negative value (an error code) if unsuccessful.

Connections are tracked so that deleting either the source or the destination instrument automatically breaks all connections to the deleted instrument.

Attachments

In addition to connecting instruments to one another, other kinds of audio folio Items may be attached to certain instruments and templates. Samples contain the digitized audio data that sample player instruments play. Envelopes define the time-variant function used to drive envelope instruments (e.g., `envelope.dsp`). These are covered in greater detail in Chapters 4 Playing a Sampled Sound and 5 Controlling Sound Parameters, respectively.

Playing Instruments

To use an instrument, you must first start it (the equivalent of plugging it in and turning it on). If it is a note-playing instrument, you can then release it so that it can finish its note. You may stop instruments to return them to their initial idle state.

Starting an Instrument

To start an instrument, use this call:

```
Err StartInstrument (Item instrument, const TagArg *tagList)
```

The call accepts two arguments: the item number of the instrument to start, and an optional tag list, which may be used to set frequency and amplitude for note-playing instruments.

When `StartInstrument()` executes, it causes the specified instrument to begin execution on the DSP. `StartInstrument()` returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

The effect of starting an instrument depends on what the instrument does once it is running. For instruments that do not play notes, such as mixers and effects instruments, starting the instrument simply runs it on the DSP so you can use it to act on other instruments (think of it as switching the instrument on). For note-playing instruments, such as sampled-sound instruments and sound-synthesis instruments, starting the instrument turns it on and starts whatever note the instrument is set up to play.

Starting a Sound-Synthesis Instrument

Starting a sound-synthesis instrument simply starts the instrument's audio signal generation, which continues until the instrument is stopped. For example, starting `sawtooth.dsp` causes the instrument to run on the DSP, where it generates a sawtooth wave output until stopped.

Starting Dependent Samples and Envelopes

Starting an instrument with one or more start-dependent attachments (to items such as samples or envelopes) starts simultaneous playback of all those attachments. How the playback continues from there depends on how loops are set within the attached samples or envelopes:

- ◆ If there is no loop, playback continues until the sample or envelope is finished.
- ◆ If there is a sustain loop but no release loop, playback loops until the instrument is released. Playback then continues to the end of the sample or envelope.

- ◆ If there is a release loop but no sustain loop, playback loops until the instrument is *stopped*. (Releasing has no effect on a release loop.)
- ◆ If there is a sustain loop *and* a release loop, playback loops in the sustain loop until the instrument is released. It then plays into the release loop, where it loops until the instrument is stopped.

Playback within each attached item continues independently of other items attached to the same instrument. So, for example, a sample attached to an instrument can loop continuously while an envelope attached to the same instrument plays through to its end and then stops.

If an instrument stops, all of its attachments also stop playing. When an attachment to an instrument stops playback, the instrument itself keeps going (unless the attachment was defined as one which stops the instrument using the `AF_ATTFFATLADYSINGS` attachment flag, see Chapter 5 Controlling Sound Parameters).

If an instrument is restarted before it has stopped, its playback starts from the beginning once again. When an instrument restarts, the Audio folio stops it very quickly and then starts it again.

Note-Playing Tag Arguments

When a note-playing instrument (e.g., `sawtooth.dsp`, `sampler_16_v1.dsp`, patches based on note-playing instruments, etc.) is started by `StartInstrument()` without tag arguments, the instrument plays using whatever frequency and amplitude settings were last applied to it (e.g., the initial settings, the last value set by a `SetKnobPart()`, the last call to `StartInstrument()`, etc.).

There are two ways to start an instrument and specify that it play at a given frequency or amplitude: you can set the instrument's frequency and amplitude knobs to the desired settings and then start the instrument. Or you can use the tag arguments supported by `StartInstrument()`, which set the frequency and amplitude knobs for you without requiring you to call `SetKnob()`. Table 3-2 lists

some of the `StartInstrument()` tag arguments which set frequency.

Table 3-2 *Some of the `StartInstrument()` Tag arguments that set frequency.*

Tag Name	Description
AF_TAG_PITCH	A MIDI pitch value from 0 to 127 that specifies the appropriate frequency set by the instrument's tuning. Default tuning is 12-tone equal-tempered tuning with 60 set to middle C. You can specify a different tuning "Tuning Instruments" on page 90. This tag also performs multi-sample selection based on sample note range, and envelope pitch-based time scaling. See "Attaching Multisamples to an Instrument" on page 53
AF_TAG_FREQUENCY_FP	Play instrument at a specific frequency in Hertz. For synthetic-sound or LFO instruments, this sets the Frequency knob to the specified value. For sample players, this adjusts the SampleRate knob to play the sample at the desired frequency. For example, to play <code>sinewave.aiff</code> at 440 Hz, set this tag to 440.
AF_TAG_SAMPLE_RATE_FP	Sample rate in Hertz to set sample player's SampleRate knob to (e.g., 22050 Hz). This applies only to sample player instruments.

Table 3-3 lists some of the `StartInstrument()` tag arguments which set amplitude.

Table 3-3 *Some of the `StartInstrument()` Tag arguments that set amplitude.*

Tag Name	Description
AF_TAG_AMPLITUDE_FP	Value to set instrument's Amplitude knob to before starting instrument. Valid range -1.0 to 1.0.

Table 3-3 Some of the `StartInstrument()` Tag arguments that set amplitude.

Tag Name	Description
AF_TAG_VELOCITY, AF_TAG_SQUARE_VELOCITY, AF_TAG_EXPONENTIAL_VELOCITY	Various mappings of MIDI key velocity to instrument amplitude. Velocity values are in the range of 0 to 127. See the description of <code>StartInstrument()</code> in the <i>3DO Audio and Music Programmer's Reference</i> for more information. These tags also perform multi-sample selection based on sample velocity range. This function is described in Chapter 4, <i>Playing a Sampled Sound</i> .

Whenever you use these tag arguments in `StartInstrument()`, they have the same effect as setting either the frequency or amplitude knob of the specified instrument. They also can be used to perform multi-sample selection and envelope time scaling. If an instrument does not have a frequency knob, then frequency tag arguments have no effect. If an instrument does not have an amplitude knob, then amplitude tag arguments have no effect. And if an instrument has a frequency or amplitude knob but has another instrument (such as `envelope.dsp`) connected to it, the knob is not available, so the tag arguments do not affect it.

Releasing an Instrument

Whenever you start a note-playing instrument that has a sustain loop, the instrument continues playing until it is either released or stopped. Releasing the instrument allows it to continue on to its release phase (either a release loop or the end); stopping the instrument simply stops playback. To release an instrument, use this call:

```
Err ReleaseInstrument (Item Instrument, const TagArg *tagList)
```

The call accepts two arguments: the item number of the instrument to be released and an optional tag list

When `ReleaseInstrument()` executes, it releases the specified instrument. It returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

Releasing an instrument that is not playing a sustain loop (either on its own or through an attachment) has no effect at all on the item. Nor, for that matter, does it have any effect on instruments that do not play notes, such as mixers.

Stopping an Instrument

Stopping an instrument makes it inactive, stopping it from running on the DSP. Stopping a note-playing instrument means that its note stops dead in its tracks; stopping other instruments, such as mixers, effects, instruments, and the like, means that they no longer accept inputs and create outputs. To stop an instrument, use this call:

```
Err StopInstrument (Item Instrument, const TagArg *tagList)
```

The call accepts the item number of the instrument to be stopped and an optional tag list. It returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

Stopping an instrument also stops playback of all attachments to that instrument. Stopping a note-playing instrument in full voice can result in a click. It is wise to apply an amplitude envelope that tapers to nothing or to turn down the amplitude completely before stopping a note-playing instrument.

Instruments are automatically stopped when they are deleted.

Note: *Although stopped instruments produce no output, they still consume DSP resources. To release the DSP resources of unused instruments, delete them with `DeleteInstrument()`.*

Cleaning Up When Finished

Deleting an Instrument

When a task finishes using an instrument, it should delete the instrument as soon as possible to free resources for other instruments. To delete an instrument, use this call:

```
Err DeleteInstrument (Item instrument)
```

The call accepts the item number of the instrument you wish to delete. When it executes, it deletes the instrument and frees its resources. It returns a non-negative value if successful, or a negative number (an error code) if unsuccessful. All instruments created from a particular template are automatically deleted when that template is deleted.

Note: *If you created an instrument with the `LoadInstrument()` call, do not use `DeleteInstrument()` to delete it. If you do, the instrument is deleted and its template is still there with no way to get rid of it. Use `UnloadInstrument()` to delete instruments created by `LoadInstrument()`.*

Deleting an Instrument Template

When a task no longer requires an instrument template for allocating more instruments, it should get rid of the template to free system resources. There are three possible calls depending on how the instrument template was created:

Just as there each kind of instrument template has its own creation method, each has its own deletion method:

- ◆ **Standard.** Standard instrument templates are deleted using `UnloadInstTemplate()`.
- ◆ **Mixers.** Mixer templates are deleted with `DeleteMixerTemplate()`.
- ◆ **Patches.** `DeletePatchTemplate()` and `UnloadPatchTemplate()` are ways to delete patch templates.

Each takes the item number of the instrument template to delete as its only argument, and returns a non-negative value if successful, or a negative number (an error code) if unsuccessful.

When you delete an instrument template, any instruments created with that template are also deleted. Be *certain* that you do not delete an instrument template when you have associated instruments still in use.

Deleting an Instrument loaded by LoadInstrument()

If you created an instrument using the convenience call `LoadInstrument()`, you should delete that instrument and its template (when finished with them) with the following call:

```
Err UnloadInstrument (Item instrument)
```

The call accepts the item number of the instrument to delete. When it executes, it deletes the specified instrument and also the instrument template used to create it. It returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

Note: *Do not use this call to delete an instrument that was created by `CreateInstrument()`. The call deletes the template when it deletes the instrument and all the other instruments associated with that template are deleted at the same time, whether you intended to delete them or not.*

Close the Audio Folio

When a task finishes using the Audio folio, it should close its connection to the folio using this call:

```
Err CloseAudioFolio (void)
```

The call accepts no arguments, returns a non-negative value if successful, and a negative number (an error code) if unsuccessful. When `CloseAudioFolio()` executes, it disconnects the task from the Audio folio.

Introducing the Audio Clock

One of the most important aspects of audio programming is timing. The audio folio provides a global *audio clock*, a timer which runs at approximately 240 Hz.

The 240 Hz rate was chosen because it is commonly used to time MIDI sequences, and because it is close to the 1/4 video frame rate of SMPTE. The Audio folio provides a set of timing calls that refer to the audio clock. The Audio folio also has a timing notification mechanism, the *cue*, which can send a timing call back to a task at the appropriate time.

The audio clock is derived from the sample clock of the DSP, and advances every 184 sample frames. When the DSP runs at its normal rate of 44,100 Hz, the audio clock tick frequency is approximately 240 Hz (239.674 to be more accurate) and each tick has a duration of approximately 4.172 milliseconds.

Note: *Do not depend on the audio clock being exactly 240Hz.*

The global audio clock keeps a running total of its ticks from the moment the audio folio is started. The beginning total is 0. The running total, which gives elapsed time in ticks, is stored as an unsigned 32-bit integer `AudioTime`. The clock wraps around when it reaches its maximum; that is, it stores up to 4,294,967,296 ticks, then jumps back to 0 and starts accumulating once again. At 240 ticks per second, 4,294,967,296 is enough ticks to measure the length of approximately 207 days.

Applications are not affected when the clock wraps around, because the Audio folio treats time as though it were circular. That is, it understands that 23 ticks is 30 ticks after 4,294,967,290 ticks. And because time is circular, a 3DO system left running as a demo in a store window will not stop every 207 days when the clock resets itself.

The audio clock's *relative* time is stored as a signed 32-bit integer. It sees the "future" as a positive value and the "past" as a negative value. The drawback to this interpretation of relative time is, if you refer to more than 103 days in the future, the clock interprets it as the past. You must cause the clock to wake up prior to the 103rd day and reset itself.

Reading the Global Audio Clock

To read the current time of the global audio clock, use the call:

```
AudioTime GetAudioTime (void)
```

The call takes no arguments and returns the 32-bit global audio clock time as an `AudioTime` value.

If you need to know the global audio clock's frequency, use this call:

```
Err GetAudioClockRate (Item clock, float32 *hertz)
```

where `clock` is set to `AF_GLOBAL_CLOCK` and `hertz` points to a `float32` variable to receive the clock rate in Hertz. For example,

```
{  
    float32 clockRate;  
  
    GetAudioClockRate (AF_GLOBAL_CLOCK, &clockRate);  
}
```

To check the current duration of a single tick of the global audio clock in DSP sample frames, use the call:

```
uint32 GetAudioDuration (void)
```

The call accepts no arguments and returns the current global clock duration in DSP frames.

Example 3-1 *Example of using the audio timer lifted from the audio example tone.c.*

```
{
    float32 TicksPerSecond;
    Item SleepCue;

    .
    .
    .

    /*
    ** Create a Cue item that we can use with the Audio Timer functions.
    ** It contains a Signal that is used to wake us up.
    */
    SleepCue = CreateCue (NULL);

    /*
    ** The audio clock rate is usually around 240 ticks per second.
    ** It is possible to change the rate using SetAudioClockRate().
    ** We can query the audio rate by calling GetAudioClockRate().
    */
    GetAudioClockRate (AF_GLOBAL_CLOCK, &TicksPerSecond);

    /*
    ** Play a note using StartInstrument.
    ** You can pass optional TagArgs to control pitch or amplitude.
    */
    StartInstrument (OscIns, NULL);

    /*
    ** Go to sleep for about 2 seconds.
    */
    SleepUntilTime (SleepCue, GetAudioTime() + (2.0 * TicksPerSecond));

    /* Now stop the sound. */
    StopInstrument (OscIns, NULL);

    .
    .
    .
}
```

Playing a Sampled Sound

This chapter contains the steps needed to play a sampled sound. It contains the following topics:

Topic	Page Number
Introduction	43
Loading and Attaching Samples	46
Example Program	55
Modifying Attachments	55
Independently Controlling Attachments	57
Linking Attachments	58

Introduction

The most convenient way to produce a sound effect is to play a digital audio sample. You can digitally record any sound you desire, a drum, a duck quack, or an explosion, and then play back that exact sound in your title. Samples can be recorded using a DAT machine, or a computer-based audio recording system. You can also purchase audio samples, or use sampled sounds from the 3DO Content Library.

A digital audio sample consists of an array of numbers corresponding to the sound's amplitude over time. A typical audio sample may contain 44100 numbers for a second of audio. This matches the 44100 sample rate of an audio CD. You can use lower sample rates to save memory but high frequency sounds like a cymbal crash will become muffled. Sample data can vary in the number of channels

(mono, stereo, etc.), the sample rate, the data compression format, and length. Portions of samples can be looped to produce a continuous tone when, for example, playing a clarinet note.

One advantage of samples is that they can accurately reproduce any sound. One disadvantage of samples is that what they reproduce is always the same. This is not realistic because ducks quack differently each time they make sounds. Samples are also difficult to use for continuous natural sounds like wind or rain because looped samples will be quite noticeable and seem artificial. For continuous sounds, or sounds that vary every time, you may wish to use synthetic Patches as described in Chapter 7.

Sampled Sound

Sampled sound instruments play back 8- and 16-bit sampled sounds. Samples may be uncompressed or compressed using a variety of methods.

Sample player names are constructed as follows:

```
sampler_COMPRESSION_{F|V}|CHANNELS.dsp
```

For example:

```
sampler_cbd2_v2.dsp = cubic, variable, stereo  
sampler_16_f1.dsp = 16 bit, fixed-rate, mono
```

Table 4-1 shows the standard sampled sound instruments currently available in the Audio folio and lists their playback characteristics.

Table 4-1 Sampled sound instruments.

Instrument Name	Sample Data Size	Sample Storage Format	Playback Sample Rate	Stereo/Mono
sampler_16_f1.dsp	16-bit	Literal	Fixed	Mono
sampler_16_v1.dsp	16-bit	Literal	Variable	Mono
sampler_16_f2.dsp	16-bit	Literal	Fixed	Stereo
sampler_16_v2.dsp	16-bit	Literal	Variable	Stereo
sampler_8_f1.dsp	8-bit	Literal	Fixed	Mono
sampler_8_f2.dsp	8-bit	Literal	Fixed	Stereo
sampler_8_v1.dsp	8-bit	Literal	Variable	Mono
sampler_8_v2.dsp	8-bit	Literal	Variable	Stereo
sampler_adp4_v1.dsp	4-bit	ADPCM Intel/DVI 4:1	Variable	Mono
sampler_cbd2_f1.dsp	8-bit	CBD2 2:1	Fixed	Mono
sampler_cbd2_f2.dsp	8-bit	CBD2 2:1	Fixed	Stereo
sampler_cbd2_v1.dsp	8-bit	CBD2 2:1	Variable	Mono
sampler_cbd2_v2.dsp	8-bit	CBD2 2:1	Variable	Stereo
sampler_drift_v1.dsp	16-bit	Literal	Variable	Mono
sampler_raw_f1.dsp	16-bit	Literal	Fixed	Mono
sampler_sqs2_f1.dsp	8-bit	SQS2	Fixed	Mono
sampler_sqs2_v1.dsp	8-bit	SQS2	Variable	Mono

Note: All 16-bit samples are big endian. Any little-endian samples will require byte swapping (e.g., a propriety 16-bit sample file format from a PC).

Sample data, used by the Audio folio, is typically stored in the AIFF or AIFC format. The data stored in an AIFC file can be stored in a compressed form, using several different compression formats. The sample formats 3DO supports are: 8-bit, 16-bit, SQS2, CBD2, and ADP4.

A sampled sound instrument's input sample size is the size of the sample it expects to read: 4 bits, 8 bits, or 16 bits. The instrument's output is always 16-bit, so if it reads 8-bit original samples it must convert them to 16-bit values. The instruments designed to read literal sample data simply add 8 less significant bits of 0s to the 8-bit value (for example, 10010111 becomes 10010111 00000000). The SQS2 and CBD2 compression formats, which are also stored as 8 bits per sample, are much better at preserving 16-bit information.

The rate at which samples are read from memory is either fixed at 44100 sample frames per second (fixed-rate sample players such as `sampler_16_f1.dsp`) or variable in the range of 0 to 88200 sample frames per second (variable-rate sample players such as `sampler_16_v1.dsp`). By playing sample data at a rate higher or lower than its original recording rate, these instruments can change the pitch of the sample so that it is higher or lower than when originally recorded. For example, using a 44,100 Hz instrument to play back a voice recording made at 22,050 Hz produces the voices an octave higher, that speak twice as fast, a cheap way to produce chipmunk voices. Variable-rate sample players perform linear interpolation to convert from the desired variable rate to the DSP's 44100 Hz internal sample rate.

Portfolio sampled sound voices come in mono and stereo varieties. Mono instruments read sample data so that all samples go in succession to a single output channel. Stereo sample players read sample data with the left and right channel samples interleaved such that alternating samples go to each of the two output parts. Within a stereo sample frame the samples are ordered: left, right.

The Music library (described in Chapter 2, "Music Library Calls," in the *3DO Music and Audio Programmer's Reference*) includes a call named `SampleItemToInsName()`. This call queries the sample to determine the sample rate, the sample's width, whether it's mono or stereo, and what its compression type is. Then, the call returns the name of an appropriate sampled sound instrument to play the sample.

Loading and Attaching Samples

A sampled-sound instrument must have a sample attached to it or it cannot play. And before a sample can be attached to a sampled sound instrument, it must first be loaded from disc into RAM or defined within RAM. The Audio folio offers sample calls to create, attach, detach, and delete samples.

Loading Samples from Disk

The Music Library provides a routine to load samples stored in either the AIFF or AIFC file format, so sampled sounds stored on disc should be stored in either of those formats. (You *can* store sampled sounds in other formats if you're willing to write your own loader and pass the sample on to the Audio folio.) The AIFF

format, which is a subset of the AIFC format, supports only uncompressed sampled sounds. An AIFC file can contain either a compressed or an uncompressed sampled sound. You can create these files using sound development tools such as AudioMedia, Alchemy, CSound, and SoundHack.

An AIFF or AIFC file contains the sample data for a sampled sound, along with other important auxiliary information such as the sample size (8 or 16 bits), recording sampling rate, looping points, and whether the sample is in stereo or mono. When the sample is loaded, it becomes a sample item. The sample data goes into the task's own memory, while the auxiliary sample information is stored as part of the item in system memory.

Simple Loading

The simplest sample loading audio call is this:

```
Item LoadSample( char *Name )
```

The call accepts a pointer (*Name) to a string with the filename of the AIFF or AIFC file that contains the sampled sound. When executed, `LoadSample()` finds the file, allocates task memory for the sample data, and loads the sample sound data there. It reads the auxiliary sample data, and creates a sample item that contains settings to match the auxiliary data. If successful, the call returns the item number of the loaded sample. If unsuccessful, it returns a negative value (an error code).

Note: To use `LoadSample()` you need to `#include (audio:parse_aiff.h)`.

Creating an Empty Sample

To create an empty sample, call the `CreateSample()` function. You must allocate memory for the sample and then fill it, using any technique you choose. This is useful if you want to buffer parts of a large AIFF sample file in memory, synthesize a sample from scratch, or use a custom decompression technique to load a sample from disc. Once created, the sample can be used like any other sample. An example of using `CreateSample()` is shown in Example 4-1.

Example 4-1 *Creating an empty sample.*

```
/*Allocate data from memory accessible to audio DMA. * /
SampleData = (int16 *) AllocMem( NumBytes, MEMTYPE_NORMAL);
If (SampleData == NULL)
{
    ERR(("Could not allocated sample data.\n"));
    goto cleanup;
}
/* Use tags to define sample format and location. */

Tags[0].ta_Tag = AF_TAG_ADDRESS;
Tags[0].ta_Arg = (int32 *) Data;
Tags[1].ta_Tag = AF_TAG_NUMBYTES;
Tags[1].ta_Arg = (int32 *) NumBytes;
Tags[2].ta_Tag = AF_TAG_CHANNELS;
Tags[2].ta_Arg = (int32 *) 2;          /* stereo, optional */
Tags[3].ta_Tag = TAG_END;
SampleItem = CreateSample (Tags);
CHECKRESULT(SampleItem, "CreateSample");
```

The tags AF_TAG_ADDRESS and either AF_TAG_FRAMES or AF_TAG_NUMBYTES are mandatory.

Caution: *If you write to the sample data memory after calling `CreateSample()`, then you MUST flush the data cache before playing that data. An example situation is when you are spooling data off of disc and into your sample buffer. Use the kernel function `WriteBackDCache()`.*

To use `CreateSample()`, you create a list of tag arguments. The possible tags are shown in Table 4-2.

Table 4-2 Tag arguments that define the characteristics of a sampled sound.

Tag Name	Description
AF_TAG_ADDRESS	Pointer to sample data. If this tag is not specified on creation, memory is allocated based on AF_TAG_NUMBYTES or AF_TAG_WIDTH.
AF_TAG_BASEFREQ_FP	The frequency of the sample when played at the original sample rate. This is an alternative to using AF_TAG_BASENOTE.
AF_TAG_BASENOTE	MIDI note number for this sample when played at the original sample rate. This defines the frequency conversion reference note for the StartInstrument() AF_TAG_PITCH tag. Default is 60.
AF_TAG_CHANNELS	Number of channels (or samples per sample frame). For stereo, this would be 2.
AF_TAG_COMPRESSIONRATIO	Compression ratio of sample data. Uncompressed data has a value of 1. Default is 1.
AF_TAG_COMPRESSIONTYPE	32-bit ID representing AIFC compression type of sample data (e.g. ID_SDX2). 0 for no compression.
AF_TAG_DELAY_LINE	Creates a delay line consisting of ta_Arg bytes (not frames). Mutually exclusive with AF_TAG_IMAGE_ADDRESS and AF_TAG_NAME.
AF_TAG_DETUNE	Amount to detune the MIDI base note in cents to reach the original pitch. Must be in the range of -100 to 100. Default is 0.
AF_TAG_FRAMES	Length of sample expressed in frames. For a stereo sample, this is the number of stereo pairs in the sample. When setting, this tag is mutually exclusive with AF_TAG_NUMBYTES.
AF_TAG_HIGHNOTE	Highest MIDI note number at which to play this sample when part of a multisample. Range is 0 to 127. Default is 127.
AF_TAG_HIGHVELOCITY	Highest MIDI note on velocity at which to play this sample when part of a multisample. Range is 0 to 127. Default is 127.
AF_TAG_LOWNOTE	Lowest MIDI note number at which to play this sample when part of a multisample. Range is 0 to 127. Default is 0.
AF_TAG_LOWVELOCITY	Lowest MIDI note on velocity at which to play this sample when part of a multisample. Range is 0 to 127. Default is 0.

Table 4-2 Tag arguments that define the characteristics of a sampled sound.

Tag Name	Description
AF_TAG_NUMBITS	Number of bits per sample (uncompressed). On creation, defaults to value from file or 16. When setting, is mutually exclusive with AF_TAG_WIDTH. Default is 16.
AF_TAG_NUMBYTES	Length of sample expressed in bytes. This tag is supported for all operations for non-delay line samples. When setting, this tag is mutually exclusive with AF_TAG_FRAMES.
AF_TAG_RELEASEBEGIN	Frame index of the first frame of the release loop (0..NumFrames-1), or -1 if no release loop. Must be used in conjunction with AF_TAG_RELEASEEND. Default is -1.
AF_TAG_RELEASEEND	Frame index of the first frame after the last frame in the release loop when (1..NumFrames), or -1 if no release loop. Must be used in conjunction with AF_TAG_RELEASEBEGIN. Default is -1.
AF_TAG_SAMPLE_RATE_FP	Original sample rate for sample expressed in float32 Hz. Default is 44100.000. The float32 value must be parsed using <code>ConvertFP_TagData()</code> .
AF_TAG_SUSTAINBEGIN	Frame index of the first frame of the sustain loop (0..NumFrames-1), or -1 if no sustain loop. Must be used in conjunction with AF_TAG_SUSTAINEND. Default is -1.
AF_TAG_SUSTAINEND	Frame index of the first frame after the last frame in the sustain loop (1..NumFrames), or -1 if no sustain loop. Must be used in conjunction with AF_TAG_SUSTAINBEGIN. Default is -1.
AF_TAG_WIDTH	Number of bytes per sample (uncompressed). When setting, is mutually exclusive with AF_TAG_NUMBITS. Default is 2.

The tag argument list is terminated with TAG_END or NULL. Any of the tags listed in Table 4-1 that you do not supply, are filled in with default values. Note that a sample frame is the set of data sent to the DSP for one DSP frame. In a stereo sample, that's two samples. A frame index is an offset from the first frame of the sample data. Thus, the first frame has a frame index of 0.

Sample Loops

Four of the sample tag arguments set loops in the sample data. A loop defines a section of the sample that is played over and over until released or stopped. Audio folio samples can contain one, two, or no loops. The two possible loops are:

- ◆ The *sustain loop* loops after a note has started and before it has been released.

- ◆ The *release loop* loops after a note has been released.

The sustain loop is defined by the two tag arguments `AF_TAG_SUSTAINBEGIN` and `AF_TAG_SUSTAINEND`. These tag arguments set the beginning and end of the loop; each is an index from the beginning sample frame of the sample data. (Index 0 falls just before the first sample frame.) If you want the sustain loop to extend from the 189th sample frame of the envelope to the 4486th sample frame, `AF_TAG_SUSTAINBEGIN` should equal 188 and `AF_TAG_SUSTAINEND` should equal 4486. (It helps to think of an index as a point between samples. Index 188 is just before the 189th sample; index 4486 is just after the 4486th sample.)

The minimum loop size is 8 bytes because of DMA restrictions.

The release loop, like the sustain loop, is defined with two sample tag arguments: `AF_TAG_RELEASEBEGIN` and `AF_TAG_RELEASEEND`. These tag arguments serve the same purpose for the release loop as the `SUSTAIN` tag arguments do for the sustain loop.

Sample Trigger Points

When you define a sample with loops, it helps to know which points of the sample play during different stages of note play. The Audio folio offers three calls that trigger different stages of a note. Those stages (along with their functions in sample playback) are

- ◆ *Start* begins playback of the sample from the first sample frame. Playback continues into the sustain loop (if defined), which loops until released. If there is no sustain loop, playback continues into the release loop. If there are no loops, playback continues to the end of the sample.
- ◆ *Release* asks that the sustain loop (if defined) finish its current iteration and that sample playback then continue beyond the sustain loop's end point into the release loop (if defined) or to the end of the sample if there is no release loop.
- ◆ *Stop* asks the sample to stop playing, no matter where playback is in the sample data.

If an instrument with a sustain loop is released before playback enters the sustain loop, the sustain loop plays only once. An instrument with a sustain loop loops indefinitely without a release or a stop, while an instrument with a release loop loops indefinitely without a stop.

Attaching a Sample to an Instrument

Once you have loaded (or defined) a sample, you must attach it to a sampled sound instrument before you can play the sample. To do so, use this call:

```
Item CreateAttachment( Item master, Item slave, const
                      TagArg *tagList )
```

The call takes three arguments: *master*, the item number of an instrument or template; *slave*, the item number of a sample or envelope; and **tagList*, a pointer to a tag list.

When `CreateAttachment()` executes, it attaches a Sample or Envelope to a particular Instrument or Template. If successful, the call returns the item number of the attachment. If unsuccessful, it returns a negative value (an error code).

Table 4-3 Tag arguments that define the characteristics of an attachment.

Tag Name	Description
AF_TAG_CLEAR_FLAGS	Accepts flag bits that it clears. These flags are the same as those used for AF_TAG_SET_FLAGS.
AF_TAG_HOOKNAME	The name of the sample or envelope hook in the instrument to attach to. For sample hooks, defaults to the first one listed for each instrument. For envelopes, defaults to "Env".
AF_TAG_MASTER	Instrument or Template item to attach envelope or sample to. Must be specified when creating an Attachment. Set for you by <code>CreateAttachment()</code>
AF_TAG_SLAVE	<u>Envelope or Sample</u> Item to attach to instrument. Exactly one of AF_TAG_ENVELOPE or AF_TAG_SAMPLE must be specified. Set for you by <code>CreateAttachment()</code>
AF_TAG_SET_FLAGS	Accepts flag bits that it turns on. These flags are AF_ATTFF_NOAUTOSTART, which sets the attachment for independent play, and AF_ATTFF_FATLADYSINGS, which sets the instrument to stop when this attachment stops. (Both these flag constants are found in <i>audio.h</i> .)
AF_TAG_START_AT	A uint32 value that specifies the point at which to start when an attachment is played. For a sample, this is an index to a sample frame. For an envelope, this is an index to an envelope point. STARTAT is not currently implemented for envelopes.

The Attachment Item

The attachment item, created by `CreateAttachment()`, is important because it defines the attachment between a sample and an instrument, both of which remain independent entities. The attachment lists the sample and the instrument and ties the two together.

Because an attachment item maintains the sample and the instrument as independent entities (it does not incorporate the sample into the instrument), one sample can be attached to more than one instrument. And more than one sample can be attached to a single FIFO of an instrument.

You can alter attachment items with attachment calls; to do so, you need the attachment item number returned by `CreateAttachment()`.

Attaching Multisamples to an Instrument

It is tempting to try to create a realistic sounding instrument by attaching a single sample to an instrument and then playing the sample back at different rates to create a full range of pitches. There is one major problem with this approach: the sample no longer sounds real once it is played back at a rate much slower or faster than the original rate. For example, a grand piano sample of middle C might sound fine played back at rates that create pitches up to an E above or an A below, but going much higher than that makes the sample sound unreasonably short and tinny, and going lower makes it sound too long and muddy.

The solution is multisampling: creating many different samples for a single instrument, one sample for each reasonable range of pitches. For example, a middle-C piano sample might be used for an instrument's A to D range, while an F above middle C on the piano may be used for the E-flat to A-flat range above, and so on, with a new sample for every half-octave of the instrument's range.

To create a multisample sound with a sampled sound instrument, you simply use `CreateAttachment()` to attach as many samples as you need to a single FIFO of the instrument. Each of the samples should be defined to have an exclusive pitch or velocity range in which it plays. Some audio editors allow you to set the pitch range for the sample file. If you cannot set the pitch range there, you can set it using the sample tag arguments `AF_TAG_LOWNOTE` and `AF_TAG_HIGHNOTE`. You can set the velocity range using the tag arguments `AF_TAG_LOWVELOCITY` and `AF_TAG_HIGHVELOCITY`. The call `SetAudioItemInfo()`, allows you to set these tag argument values for any existing sample item. You can also set the record pitch of each multisample using `AF_TAG_BASENOTE` and `AF_TAG_DETUNE`.

When you play the instrument later, specifying a pitch or a velocity, the Audio folio plays only the sample that is set to play in that pitch and velocity range. If more than one sample fits the pitch and velocity criteria, the Audio folio plays the first of the appropriate samples connected to the instrument. If no sample fits the pitch and velocity criteria, no sample plays.

Note that all samples in a multisample configuration must have the same format: the same number of bytes per sample, the same number of channels (stereo or mono), and the same compression type.

You can use multisamples with non-variable-rate instruments such as `sampler_16_f1.dsp`. The pitch will not affect the frequency but it can be used to select a sample. This can be used to put a drum kit on a single instrument.

Detaching a Sample from an Instrument

When a task finishes playing a sample through a sampled-sound instrument, it should detach the sample from the instrument. To detach an instrument, use this call:

```
Err DeleteAttachment ( Item Attachment )
```

The call accepts the item number of the attachment that connects the sample to the instrument. When it executes, the call deletes the attachment item, detaching the sample from the instrument. `DeleteAttachment()` returns 0 if successful, or a negative value (an error code) if unsuccessful.

Deleting a Sample

Once a task is finished with a sample, it should delete the sample to free system resources and task memory. To do so, use this call:

```
Err DeleteSample( Item Sample )
```

The call accepts the item number of the sample to be deleted. When it executes, `DeleteSample()` deletes the sample item, and deletes any attachments to the sample. It returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

If the Sample was created with `{ AF_TAG_AUTO_FREE_DATA, TRUE }`, then the sample data is also freed when the Sample Item is deleted. Otherwise, the sample data is not automatically freed.

Debugging a Sample

If you are having trouble working with a sample in your code and would like to see *all* the available information about the sample item, the development version of Portfolio offers this call:

```
Err DebugSample( Item Sample )
```


The call accepts the item number of the sample for which you want information. When it executes, it prints sample information on the screen of the Macintosh connected to your development system. If successful, it returns 0. If unsuccessful, it returns a negative value (an error code).

Please note that this call is for debugging only. It will *not* be available in the production version of Portfolio, so do not use it to get information for your task to act on. Use `GetAudioItemInfo()` instead.

Example Program

See *Examples/Audio/Misc/Playsample.c* on the 3DO Portfolio and Toolkit, V 2.0 CD-ROM for a demonstration of how to load and play an AIFF file.

Modifying Attachments

Whenever you attach an envelope or a sample to an instrument, you create an attachment item that controls the attachment's attributes. These attributes include the item numbers of the instrument, the item number of the envelope or sample being attached, and more. By modifying the attachment's attributes, you can control the way the attachment plays when the instrument is played. By linking two or more attachments together with a linking audio call, you can set a sequence of attachments to play back during a note.

Setting an Attachment's Attributes

To see the current settings of an attachment, use the `GetAudioItemInfo()` call. You can use the attachment tag arguments described in Table 4-3 on page 52 to define what information you want to see.

To change any of the attachment's attributes, use these same tag arguments with the `SetAudioItemInfo()` call.

Specifying a Start Point

When an attached sample or envelope starts to play a note, it starts by default with the first sample in the sample data or with the first point of the envelope point set. If you would like the sample or envelope to start at a later point, set the tag argument `AF_TAG_START_AT` to a value greater than 0 (the default value).

Note: Do not set `AF_TAG_START_AT` past a *SustainEnd* loop marker.

The value you set is an index into the sample or envelope. If used for a sample, the value sets the sample frame where sample playback starts. For example, if you set `AF_TAG_START_AT` to 32755 for a stereo sample, then you specify the 32,756th pair of samples in the sample data (the first sample is 0). If used for an

envelope, the value sets the envelope point where envelope playback starts. For example, if you set `AF_TAG_START_AT` to 5, then you specify the sixth point in the envelope.

Note: *The Audio folio does not currently support specifying an envelope start point.*

Setting a Start-Independent Attachment

When you start an instrument with the `StartInstrument()` call, all components of the instrument start playing at the same time: the synthesized waveform, the attached envelope (or envelopes), the attached sample, and so on. There can be times, however, when you want an attached envelope or sample to start independently of the instrument. For example, you can set up a sampled-sound instrument to play the sample of a murmuring crowd. You can then attach an envelope that swells the amplitude for a second and then drops it back to a murmur. A sustain loop keeps the crowd murmuring when the instrument plays. If the envelope is triggered independently, then it can play whenever the program wants the intensity of the crowd noise to step up for a short time.

To set a start-independent attachment, set the `AF_ATT_NOAUTOSTART` flag of the `AF_TAG_SET_FLAGS` tag argument. (The flag constants are defined in the *audio.h* header file.) As long as the flag is set, the sample or envelope specified in the attachment will not start when the instrument starts; it will start only when the `StartAttachment()` call tells the attachment to start.

Note that a start-independent attachment is independent only in its start and release. It is still attached to the instrument, and will stop if the instrument stops.

Creating an Instrument-Stopping Attachment

Instruments with loops can play indefinitely once started. For example, an instrument with a sustain loop can play and play while it waits for a release or a stop. Or an instrument with a release loop can play and play while it waits for a stop. You can always stop a looping instrument with a stop call, but you may want to time the stop to occur simultaneously with the end of an envelope or a sample. For example, a sample attached to a sampled-sound instrument can be set to loop indefinitely in a release loop while an envelope plays back. The envelope shapes the sample sound amplitude. When the envelope finishes, reducing the amplitude to 0, the instrument should stop so that it will not waste DSP cycles producing inaudible sound.

The `AF_ATT_FATLADYSINGS` flag of the `AF_TAG_SET_FLAGS` tag argument, if set, specifies that the sample or envelope attached will stop the instrument at the same moment the sample or envelope itself stops. This kind of attachment is called a *stop-linked attachment*. If the flag is not set, the attached sample or

envelope is not a stop-linked attachment and can stop without any effect on the instrument's playback. Note that when the instrument stops, all attachments stop with it whether they are finished or not.

Independently Controlling Attachments

When an instrument has start dependent attachments to samples and envelopes, those attachments start, release, and stop when the instrument does. If, however, an instrument has start independent attachments to items, then those attachments are not affected by instrument start and release calls. The Audio folio supplies separate calls to start, release, and stop start independent attachments. You can use these same calls to start, release, and stop start dependent attachments as well.

Starting an Attachment

To start an attachment, use this call:

```
Err StartAttachment( Item Attachment, TagArg *tp )
```

The call accepts the item number of an attachment and a pointer to a list of tag arguments. The tag arguments are not currently defined for this call. When `StartAttachment()` executes, it starts playback of the item listed in the specified attachment: a sample or an envelope. If successful, the call returns 0; if unsuccessful, it returns a negative value (an error code).

Starting an attachment is not like starting an instrument. `StartAttachment()` just triggers the playback of an attached item to start; it does not load a DSP program and start it running the way `StartInstrument()` does. The item started, however, plays back just as it would if it were started with the instrument. Its sustain and release loops work just as they do with `StartInstrument()`.

Releasing an Attachment

To release an attachment, use this call:

```
Err ReleaseAttachment( Item Attachment, TagArg *tp )
```

The call accepts the item number of an attachment and a pointer to a list of tag arguments. The tag arguments are not currently defined for this call. When `ReleaseAttachment()` executes, it releases playback of the item listed in the specified attachment so playback can continue to the release loop or (if there is no release loop) to the end of the item. If successful, the call returns 0; if unsuccessful, it returns a negative value (an error code).

Stopping an Attachment

To stop an attachment, use this call:

```
Err StopAttachment( Item Attachment, TagArg *tp )
```

The call accepts the item number of an attachment and a pointer to a list of tag arguments. The tag arguments are not currently defined for this call. When `StopAttachment()` executes, it completely stops playback of the item listed in the specified attachment. If successful, the call returns 0; if unsuccessful, it returns a negative value (an error code).

If the attachment stopped is a stop linked attachment, the instrument to which it is attached stops as well. Stopping a non-stop-linked attachment does *not* stop the instrument.

Stopping an instrument directly stops all of the instruments attachments with it.

Linking Attachments

There are times when it is more economical and flexible to work with a series of small samples instead of a single large sample. You can assemble the small samples in different orders for different situations. For example, you may have the sound of a car door opening and closing, the sound of a seat belt ratchet, the sound of seat belt buckle, and the sound of the turn of a key click. Played in one sequence, they provide the proper sounds for entering a car and starting it. Played in another sequence, they provide the proper sounds for turning off a car and leaving it.

The Audio folio allows you to link attached items so that you can play back a series of samples (or envelopes) by starting a single instrument. To link attached items, use this call:

```
Err LinkAttachments( Item At1, Item At2 )
```

The call accepts the item number of a first attachment and the item number of a second attachment. When it executes, it links the end of the first attachment to the beginning of the second attachment. It returns 0 if successful, or a negative value (an error code) if unsuccessful.

Note: `LinkAttachments()` *does not link envelopes.*

`LinkAttachments()` can be used many times to create a chain of linked attached items. When the first attached item in a linked chain starts playing, it continues playing until its end, when the second attached item starts playing, and so on to the end of last attached item in the chain. Note that sustain loops in linked attached items can hold playback in one place for a while. For example, a sustain loop in the first attached item can loop continuously until the attachment is released, at which point the attached item plays through to its end and into the

beginning of the next attached item. If there is a sustain loop in that item, playback will hold there until playback is released once again. Release loops in linked attachments are ignored.

You can use `LinkAttachments()` to create circularly linked attachments. This is helpful for creating sound buffers for spooling sound off a disc. While one sample in the linked group plays, another sample is loaded from disc. The `MonitorAttachment()` call allows the task to see where playback is located so it can plan spooling appropriately. The Music library sound file routines (described in Chapter 2, "Music Library Calls," of the *3DO Music and Audio Programmer's Reference*) use this technique.

See the `Examples/Audio/Misc/ta_attach.c` example on the *3DO M2 Portfolio and Toolkit V2.0* CD for demonstrations of how to work with attachments.

Controlling Sound Parameters

This chapter describes how to control the parameters of a sound. It contains the following topics:

Topic	Page Number
Introduction	61
Digital Signal Processing Signal Types	61
Handling Knobs	67
Creating and Attaching Envelopes	74

Introduction

The term DSP stands for “Digital Signal Processor”. Beyond merely playing samples, the 3DO DSP has the power to perform a large amount of real-time dynamic control of sound parameters (e.g., frequency, amplitude, timbre, mix, etc.). There are a variety of ways to control sound parameters, which include directly setting knob values, using envelopes, and using the output of control signal instruments.

Digital Signal Processing Signal Types

Audio and Control Signals

The 3DO DSP can process a number of different types of signals. The most common signal type is an audio signal that corresponds to a sound. If an audio signal is sent to the DAC then it can be heard through loudspeakers. For a signal

to be heard, it must have frequency components between 40 and 20000 Hertz. The actual audible frequency range depends largely on your age and species. An example of an audio signal is the output of a `sawtooth.dsp` oscillator.

These audio signals are composed of a stream of numbers, or *samples*, whose values are floating point numbers ranging from -1.0 to 1.0. This is the same range as a sine wave and is typical of DSP signals. Audio signals outside that range cannot exist in the DSP and will be clipped to a value between -1.0 and 1.0. These basic audio signals are referred to as `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`.

The DSP also generates signals that are not meant to be heard. These are called *control signals* and are used to control various aspects of another sound. Control signals are typically more slowly changing than audio signals. An example of this would be the output of the `triangle_lfo.dsp`. The term "lfo" stands for "Low Frequency Oscillator". An LFO is typically used to provide vibrato or tremolo modulation effects like in a police siren. Control signals can be either signed or unsigned. The signed control signals range from -1.0 to 1.0. The unsigned control signals range from 0.0 to 2.0 and are called `AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED`.

Other than their range, there is nothing intrinsically different between audio signals and control signals. The difference is mainly a matter of how the signal is used. You can use an audio signal as a control signal if you wish. An example of this would be FM where one oscillator has its frequency modulated by the output of another. Think of this as an extremely fast vibrato that generates complex timbres. You can also use what is normally a control signal as an audio signal if it is signed, and if it has frequencies within the range of hearing.

Signal Flow Between Connected Instruments

The Audio Folio allows you to connect the output ports of one instrument to the input or control ports of another. An example of this would be the use of a `triangle_lfo.dsp` instrument to control the amplitude of a `sawtooth.dsp` instrument. You can accomplish this by connecting the Output port of the triangle LFO to the Amplitude control port, or knob, of the sawtooth oscillator. The call would look something like this:

```
ConnectInstruments (TriLFO, "Output", SawOsc, "Amplitude");
```

As the output of the triangle LFO slowly goes back and forth from -1.0 to 1.0, the amplitude of the sawtooth varies. This is implemented as a multiplication of the two signals in the DSP. When the output of the triangle LFO passed through zero, the sawtooth oscillator would become silent.

Technical note: When the amplitude goes negative, between -1.0 and 0.0, the output of the sawtooth oscillator would be inverted. Phase inversion is difficult to hear but can be important when signals are added together. Two sawtooth waves, one normal, one inverted, may sound identical but if you add them together they can cancel each other out and result in silence. $1+(-1)=0$.

Control Signal Arithmetic

In order to precisely control the effects of a control signal, it is sometimes necessary to perform arithmetic on them. Suppose for example, that instead of having the sawtooth amplitude vary from the extremes of -1.0 to 1.0, that you wanted it to vary from 0.4 to 0.6. Thus it would always be heard but would just vary slightly in amplitude. You can accomplish this by using one of the arithmetic instruments. We can multiply the triangle LFO signal by 0.1 so that it varies from -0.1 to +0.1. Then we can add 0.5 so that the result varies between 0.4 and 0.6.

$$\text{SawtoothAmplitude} = (0.1 * \text{TriangleLFO}) + 0.5$$

$$0.4 = (0.1 * (-1.0)) + 0.5$$

$$0.6 = (0.1 * (+1.0)) + 0.5$$

We can perform this arithmetic using the `timesplus.dsp` instrument. It has three inputs, `InputA`, `InputB`, and `InputC`. The Output is `InputA*InputB+InputC`. For our example, we would connect the Output of the `triangle.dsp` to `InputA`, set `InputB` to 0.1, and set `InputC` to 0.5. The Output of the `timesplus.dsp` instrument would then be connected to the Amplitude of the `sawtooth.dsp` instrument. See Figure 5-1.

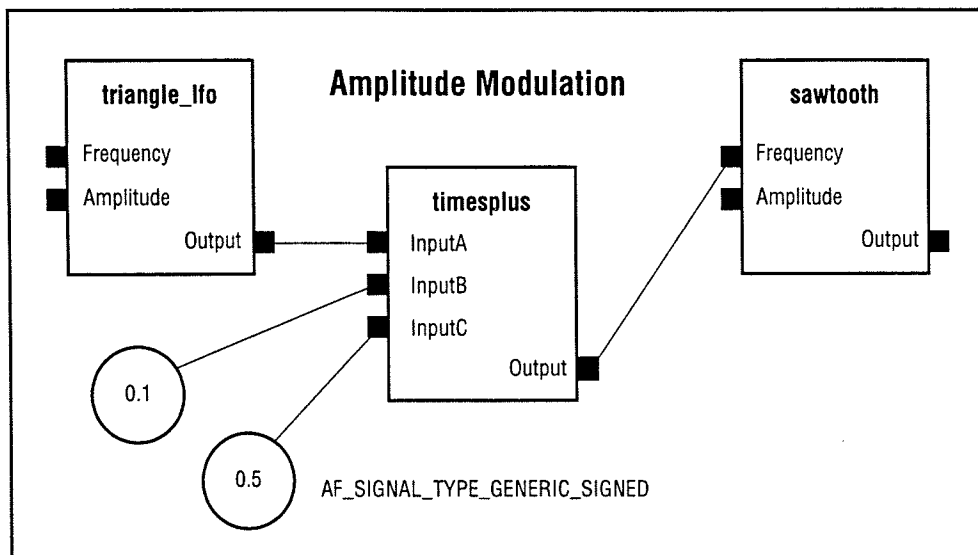


Figure 5-1 Amplitude modulation.

Knobs and Probes

The control ports on an instrument can be controlled two ways. One is by connecting the output of another instrument to it, as we have seen. The other is to create a Knob item, which allows control directly by the application. The value of the Knob can be set by calling `SetKnobPart()`. When an application is controlling a knob, it is often convenient to use units other than the generic signal values. When setting a Frequency knob for example, it would be convenient be able to set the frequency as a value in Hertz, (cycles per second). Frequency knobs, therefore, are defined as having special signal types called either `AUDIO_SIGNAL_TYPE_OSC_FREQ` or `AUDIO_SIGNAL_TYPE_LFO_FREQ`. The signal type is used to convert from real world values like Hertz to generic values for the DSP.

A Probe Item allows an application to read the current value of an instrument's output port. Probes also have signal types, which permits reading values in convenient units. Probes are discussed in more detail in *Creating the Probe and "Using Probes"* on page 89

It is important to note that the DSP only operates on generic signal types. The other signal types are only meaningful in the context of a `SetKnobPart()`, `ReadKnobPart()`, or `ReadProbePart()` call. If you connect a `triangle_lfo.dsp` to the Frequency knob of a `sawtooth.dsp` then you will be controlling the sawtooth frequency with a generic value ranging over the full

range from -1.0 to 1.0. This translates to a full frequency range of -22050.0 to 22050.0 Hertz for the sawtooth oscillator. The formula for translating between generic signals and AUDIO_SIGNAL_TYPE_OSC_FREQ is:

$$\begin{aligned}\text{Frequency(Hertz)} &= \text{Generic} * \text{SampleRate} / 2.0 \\ &= \text{Generic} * 44100.0 / 2.0 \\ &= \text{Generic} * 22050.0\end{aligned}$$

Thus a generic signal of 0.02 applied to a Frequency knob of type AUDIO_SIGNAL_TYPE_OSC_FREQ will result in a frequency of 441.0 Hz.

The following are the currently defined signal types, their ranges, and conversion formulas (where appropriate). The reference section for each standard instrument includes the signal type for each port. See the --Audio-Signal-Types-- in the *3DO Music and Audio Programmer's Reference* for more detailed information regarding the signals types.

AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

- ◆ Signed generic signal (e.g., most audio signals, amplitude).
- ◆ Range: -1.0 to 1.0

AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED

- ◆ Unsigned generic signal.
- ◆ Range: 0.0 to 2.0

AUDIO_SIGNAL_TYPE_OSC_FREQ

- ◆ Oscillator frequency in Hertz.
- ◆ Range: -22050.0 to 22050.0 Hz
- ◆ $\text{Freq (Hz)} = \text{Generic} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0)$

AUDIO_SIGNAL_TYPE_LFO_FREQ

- ◆ Low-frequency oscillator frequency in Hertz.
- ◆ Range: -86.1 to 86.1 Hz
- ◆ $\text{Freq (Hz)} = \text{Generic} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0 * 256)$

AUDIO_SIGNAL_TYPE_SAMPLE_RATE

- ◆ Sample rate in samples/second.
- ◆ Range: 0.0 to 88100.0 samples/second
- ◆ $\text{SampleRate (samp/s)} = \text{Generic} * \text{SystemSampleRate} / \text{CalcRateDivision}$

AUDIO_SIGNAL_TYPE_WHOLE_NUMBER

- ◆ Whole numbers (e.g., ..., -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, ...).
- ◆ Range: -32768.0 to 32767.0
- ◆ $\text{Whole} = \text{Generic} * 32768.0$

Note: Actual frequency ranges and precision are determined by instrument execution rate, which is determined by DAC sample rate (normally 44100) and instrument calculation rate division. As presented, the instrument execution rate is assumed to be 44100 times/second.

Type Casting for Knobs and Probes

It is quite common to perform arithmetic on control signals that are used to control frequency. Consider the earlier example of using a `timesplus.dsp` instrument to scale the output of a `triangle_lfo.dsp`. Imagine that we connected the Output of the `timesplus.dsp` to the Frequency knob of the `sawtooth.dsp` instead of the Amplitude knob. In that case, InputC would control the center frequency of the modulation, and InputB would control the frequency range. See Figure 5-1.

It would be nice if both of these could be controlled using units of Hertz, as shown in Figure 5-2. Note, however, that the inputs of `timesplus.dsp` are signals of type `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`. When we create the Knobs for them, however, we can specify that the knobs be considered to be of type `AUDIO_SIGNAL_TYPE_OSC_FREQ`.

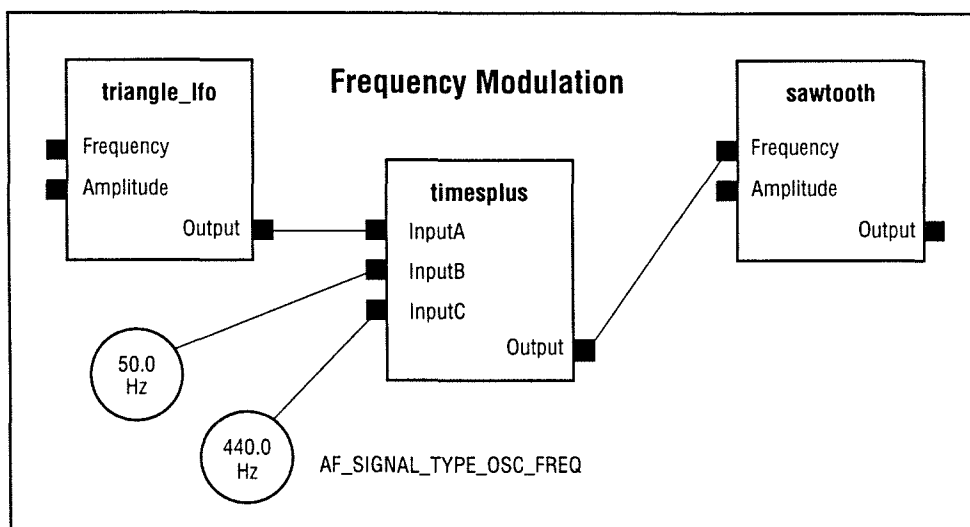


Figure 5-2 Frequency modulation.

`CreateKnob()` and `CreateProbe()` support `AF_TAG_TYPE`, which lets you override the default signal type, (similar to a type cast in 'C'). Here is how we would do that (error handling omitted for brevity):

```
modRangeKnob = CreateKnobVA (TimesPlusIns, "InputB",  
    AF_TAG_TYPE, AUDIO_SIGNAL_TYPE_OSC_FREQ,  
    TAG_END);  
SetKnob (modRangeKnob, 50.0);  
  
centerFreqKnob = CreateKnobVA (TimesPlusIns, "InputC",  
    AF_TAG_TYPE, AUDIO_SIGNAL_TYPE_OSC_FREQ,  
    TAG_END);  
SetKnob (centerFreqKnob, 440.0);
```

Handling Knobs

An instrument's knobs control its operating attributes such as frequency, mix, amplitude, and timbre. One instrument's output can control a knob's value. A task can create a Knob item, which permits the task to directly control the knob port.

A knob's setting can be changed dynamically while the instrument plays a note, which can change the sound of the note during playback, or a knob's setting can be changed statically, that is, set before note playback to set a value that affects the overall sound of the note. If you plan to set a knob dynamically, the task can do so directly, or you can connect an instrument such as `envelope.dsp` to the knob. The instrument can smoothly change the knob using the preset values stored in the envelope points. If you simply need to set a static knob value, then you can have your task create the knob and set it.

Finding Knobs

To specify an instrument's knob, you must first know the knob's name. The listing of each standard instrument's knobs appears in Chapter 3, "Instrument Templates," of the *3DO Music and Audio Programmer's Reference*. Mixers also have standard sets of knobs, which are described in the reference page Mixer of the *3DO Music and Audio Programmer's Reference*. The knob list of a patch is entirely up to the patch designer.

You can also query Instrument and Template Items about their ports (i.e., inputs, outputs, knobs, FIFOs, etc.) using the `InstrumentPortInfo` function family.

Getting Port Information by Name

To get information about a port for which you know the name, call

```
GetInstrumentPortInfoByName (  
    InstrumentPortInfo *info, uint32 infoSize,  
    Item insOrTemplate, const char *portName)
```

This function fills out an `InstrumentPortInfo` structure, which you supply. The `InstrumentPortInfo` structure members are described in Table 5-1.

Table 5-1 `InstrumentPortInfo` members.

Member Name	Description
<code>pinfo_Name</code>	Name of the port.
<code>pinfo_Type</code>	Type of the port (an <code>AF_PORT_TYPE_</code> value, e.g., <code>AF_PORT_TYPE_INPUT</code> , <code>AF_PORT_TYPE_KNOB</code>).
<code>pinfo_SignalType</code>	Signal type of the port (an <code>AUDIO_SIGNAL_TYPE_</code> value). This only applies to the following port types: <code>AF_PORT_TYPE_INPUT</code> , <code>AF_PORT_TYPE_OUTPUT</code> , <code>AF_PORT_TYPE_KNOB</code> , and <code>AF_PORT_TYPE_ENVELOPE</code> .
<code>pinfo_NumParts</code>	Number of parts that the port has. This is in the range of 1..N. Part numbers used to refer to this port are in the range of 0..N-1. This only applies to the following port types: <code>AF_PORT_TYPE_INPUT</code> , <code>AF_PORT_TYPE_OUTPUT</code> , and <code>AF_PORT_TYPE_KNOB</code> .

Here is an example which prints information about an instrument's Frequency knob (error checking omitted for brevity):

```
{  
    InstrumentPortInfo info;  
  
    GetInstrumentPortInfoByName (&info, sizeof info,  
        sawtoothins, "Frequency");  
    printf ("name='%s' port type=%d parts=%d signal type=%d\n",  
        info.pinfo_Name, info.pinfo_Type, info.pinfo_NumParts,  
        info.pinfo_SignalType);  
}
```

Getting an Instrument's Port List

You can also examine the entire port list of an instrument using the functions `GetNumInstrumentPorts()` and `GetInstrumentPortInfoByIndex()`. The first function,

```
int32 GetNumInstrumentPorts (Item insOrTemplate)
```

returns the number of ports belonging to an instrument or template item. The second function,

```
Err GetInstrumentPortInfoByIndex (  
    InstrumentPortInfo *info, uint32 infoSize,  
    Item insOrTemplate, uint32 portIndex)
```

fills out an `InstrumentPortInfo` structure just like `GetInstrumentPortInfoByName()`. Instead of supplying the name of the port, you supply the index of the port in the range of 0 to `numPorts-1`, where `numPorts` is returned by `GetNumInstrumentPorts()`. For example, here is a function which lists the ports belonging to the instrument or template `Item insOrTemplate`:

```
void DumpInstrumentPortInfo (Item insOrTemplate)  
{  
    InstrumentPortInfo info;  
    const int32 numPorts = GetNumInstrumentPorts (insOrTemplate);  
    int32 i;  
  
    printf ("Instrument 0x%x has %d port(s)\n", insOrTemplate,  
        numPorts);  
  
    for (i=0; i<numPorts; i++) {  
        if (GetInstrumentPortInfoByIndex (&info, sizeof info,  
            insOrTemplate, i) >= 0) {  
  
            printf ("%2d: '%s' port type=%d parts=%d\n",  
                i, info.pinfo_Name, info.pinfo_Type,  
                info.pinfo_NumParts);  
        }  
    }  
}
```

Creating a Knob

Before a task can set an instrument's knob, it must first create a knob item for the knob port by calling the `CreateKnob()` function:

```
Item CreateKnob (Item instrument, const char *name,  
    const TagArg *tagList)
```

The `CreateKnob()` function takes three arguments: the item number of the instrument, a string containing the name of the knob the task wants to control, and an optional list of `TagArgs`.

Table 5-2 `CreateKnob()` tag arguments.

Tag Name	Description
AF_TAG_TYPE	AUDIO_SIGNAL_TYPE_ constant indicating the signal type to use for the knob (i.e., the units used by <code>SetKnobPart()</code> and <code>ReadKnobPart()</code>). Defaults to the default signal type of the knob (for standard DSP instruments, this is described in the instrument documentation).

If the `CreateKnob()` call is successful, the call creates a knob item connecting the knob and the task, and returns its item number. If unsuccessful, it returns a negative value (an error code).

Finding Knob Parameters

Once you have created a knob, you can use its item number to find various parameters of the knob with the `GetAudioItemInfo()` call using tags described in *3DO Music and Audio Programmer's Reference*.

Table 5-3 Knob query tag arguments.

Tag Name	Description
AF_TAG_MIN_FP	The minimum setting of the knob.
AF_TAG_MAX_FP	The maximum setting of the knob.
AF_TAG_NAME	The name of the knob.

Note: The minimum and maximum values returned `AF_TAG_MIN_FP` and `AF_TAG_MAX_FP`, respectively, take into account the instrument's calculation rate.

Setting Knob Values

To set the value of a knob, use the function:

```
Err SetKnobPart (Item knob, int32 partNum, float32 value)
```


The call accepts the item number of the knob to be set, which part of a multi-part knob to set (0 for single-part knobs), and a floating-point value to which to set the knob. When executed, `SetKnobPart()` sets the knob to a specified value, clipping the value if it is too high or too low for the knob. The call returns a non-negative value if successful, or a negative value (an error code) if unsuccessful.

For single-part knobs, you may instead use the convenience function:

```
Err SetKnob (Item knob, float32 value)
```

The types and ranges of values accepted by standard instrument knobs are listed under the instrument's description in Chapter 3, "Instrument Templates," in the *3DO Music and Audio Programmer's Reference*. Frequency values are usually in Hertz (cycles per seconds); amplitude values are usually a value from 0.0 (silence) to 1.0 (full loudness).

Note: *You can also use negative amplitude values to invert a signal.*

Reading Knob Values

To read the current value of a knob part, call the function:

```
Err ReadKnobPart (Item knob, int32 partNum, float32 *valuePtr)
```

This call accepts three arguments. The first, `knob`, is the item number of the knob to be read; the second argument, `partNum`, indicates which part of a multi-part knob to read (0 for single-part knobs); the third, `valuePtr`, is a pointer to a variable to receive the floating-point value from the knob. When `ReadKnobPart()` executes, it reads the current value of the knob part, and stores this value in `*valuePtr`. If successful, it returns a non-negative value. If unsuccessful, it returns a negative value (an error code).

For single-part knobs, you may instead use the convenience function:

```
Err ReadKnob (Item knob, float32 *valuePtr)
```

Controlling Knobs with Generic Values

When `SetKnobPart()` sets a knob to a new value, it accepts the value in convenient units such as Hertz for frequency. `SetKnobPart()` then converts that value into an internal signal that the knob can actually use, a *generic* signal. The two generic signal types are `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED` (-1.0 to 1.0) and `AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED` (0.0 to 2.0).

Since instrument connections carry only generic signals, it is important to understand how generic signals affect knobs with non-generic signal types. This is described in the following material.

Sound-Synthesis Instruments

For a sound synthesis instrument, such as a `sawtooth.dsp`, values assigned to its `Frequency` knob are converted from `AUDIO_SIGNAL_TYPE_OSC_FREQ` to signed generic values measured in phase increments. A phase increment is the amount of phase increase in the waveform during one sample frame. To understand phase increments, consider that a sound synthesis instrument generates a repeating waveform (as shown in Figure 5-3). A single occurrence of the waveform is measured from 0.0 at the beginning of the waveform to 2.0 at the beginning of the next cycle of the waveform. (In traditional sound synthesis, this interval is usually measured from 0 to 360 degrees, but computers use powers of two.)

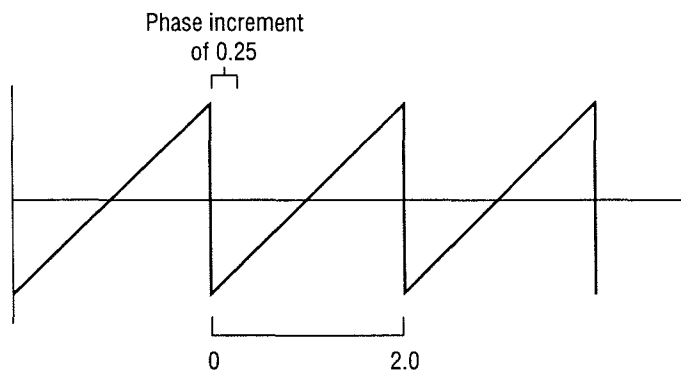


Figure 5-3 *Measuring a repeating waveform*

A waveform played back with a smaller phase increment value has a lower frequency than the same waveform played back with a larger phase increment value.

The following formula derives frequency in Hertz from sound synthesis phase increment:

$$\text{Frequency} = \text{PhaseIncrement} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0)$$

And this formula derives sound synthesis phase increment from frequency:

$$\text{PhaseIncrement} = \text{Frequency} * \text{CalcRateDivision} * 2.0 / \text{SystemSampleRate}$$

`PhaseIncrement` is allowed to be in the range of -1.0 to 1.0. For `sawtooth.dsp`, a negative phase increment causes the waveform to be a falling instead of a rising sawtooth wave.

LFO Instruments

LFO instruments, such as `triangle_lfo.dsp`, have their Frequency knobs converted from `AUDIO_SIGNAL_TYPE_LFO_FREQ` to phase increments. LFO instrument phase increments are 256 times as fine as sound synthesis phase increments, however, which gives LFOs much more precise low frequency control.

The following formula derives frequency in Hertz from LFO phase increment:

$$\text{Frequency} = \text{PhaseIncrement} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0 * 256)$$

And this formula derives LFO phase increment from frequency:

$$\text{PhaseIncrement} = \text{Frequency} * \text{CalcRateDivision} * 2.0 * 256 / \text{SystemSampleRate}$$

LFO phase increment values are also limited to the range of -1.0 to 1.0. Because of the increased precision, LFO frequency range is 256 times narrower than the sound-synthesis frequency range.

Sampled-Sound Instruments

Variable-rate sample players have a `SampleRate` knob, which has the signal type `AUDIO_SIGNAL_TYPE_SAMPLE_RATE`. These knobs are also converted to a phase increment, but it is different from those discussed so far. A setting of 1.0 plays an attached sample back at one sample per frame. Doubling that value doubles the sample rate (raising pitch of sample by an octave); halving the value halves the sample rate (lowering pitch of sample by an octave). The range for phase increments is 0.0 to 2.0, so you can raise a sample by almost a full octave or drop it so many octaves that it is no longer within the audible frequency range.

The following formula derives sample rate in samples/second from sample rate phase increment:

$$\text{SampleRate} = \text{PhaseIncrement} * \text{SystemSampleRate} / \text{CalcRateDivision}$$

And this formula derives sample rate phase increment from samples/second:

$$\text{PhaseIncrement} = \text{SampleRate} * \text{CalcRateDivision} / \text{SystemSampleRate}$$

For example, `SampleRate` of 22,050 samples/second corresponds to `PhaseIncrement` of 0.5, `SampleRate` of 11,025 samples/second corresponds to `PhaseIncrement` of 0.25.

Deleting a Knob

Once a task is finished using a Knob Item, delete it with the `DeleteKnob()` function:

```
Err DeleteKnob (Item knob)
```

The `DeleteKnob()` function takes a single argument, `knob`, that is the item number of the knob to delete. When `DeleteKnob()` executes, it deletes the knob. If successful, it returns a non-negative value. If the call is unsuccessful, it returns a negative value (an error code).

Also, all Knob Items for an Instrument Item are automatically deleted when that Instrument Item is deleted.

Creating and Attaching Envelopes

An Envelope is data that sets the shape of an audio parameter plotted over time. It typically controls one of an instrument's attributes such as its amplitude, timbre, or frequency. The Audio folio creates envelopes as envelope items. Like Sample Items may be attached to sample player instruments, Envelope items can be attached to an envelope player instruments. Envelope players read the envelope data and apply it to an instrument attribute during note play. Also like sample players, envelope players may be inside a patch or stand-alone instruments.

The standard instrument, `envelope.dsp`, is nothing *but* an envelope player. It has no sound synthesis or sampled sound source, so when it plays an envelope, it does not directly control sound attributes. Instead, it puts out a control signal through its output. The signal can be connected to a knob of any other instrument, which applies an envelope to the attribute controlled by the knob. For example, if the output of `envelope.dsp` is attached to the amplitude knob of `sampler16_v1.dsp`, then when `envelope.dsp` plays an envelope, the envelope shapes the amplitude of the audio signal coming from `sampler16_v1.dsp`. This flexibility allows you to apply envelopes to different attributes of many instruments without built-in envelope players.

Envelope Properties

An envelope item has many different properties set by tag arguments, which are described in the following sections.

Envelope Points

The tag arguments `AF_TAG_ADDRESS` and `AF_TAG_FRAMES` point to an array of points that define the envelope and specify the number of points. Each point gives a time from the beginning of the envelope and a value at that time. In series, the points determine the shape of the envelope. An example is shown in Figure 5-4.

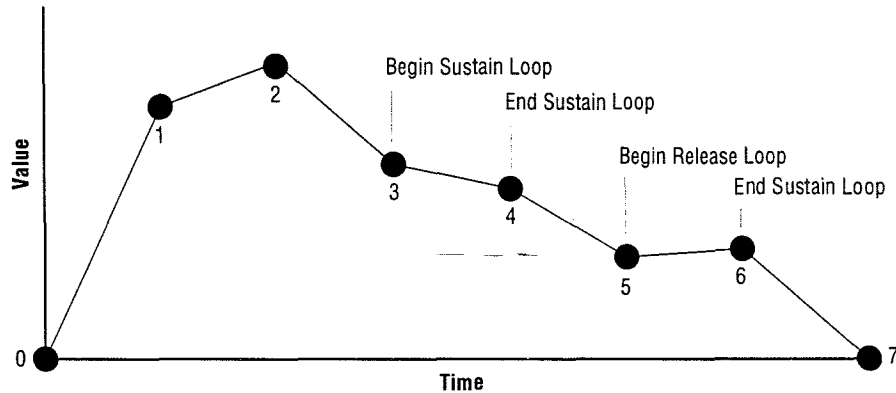


Figure 5-4 An example envelope shape

An Audio folio envelope can have as many points as you care to set, which allows you to define very complex envelope shapes. The envelope shape is defined by an array of `EnvelopeSegment`s, which is defined in `<:audio:audio.h>` as follows:

```
typedef struct EnvelopeSegment
{
    float32 envs_Value;
    float32 envs_Duration;
} EnvelopeSegment;
```

Table 5-4 `EnvelopeSegment` members.

Member Name	Description
<code>envs_Value</code>	Starting value of this segment of the array. The units and range of values are determined by the signal type of the Envelope specified by <code>AF_TAG_TYPE</code> . By default, a signed signal in the range of -1.0 to 1.0.
<code>envs_Duration</code>	Time in seconds to reach the <code>envs_Value</code> of the next <code>EnvelopeSegment</code> in the array.

Table 5-5 Tags associated with the Envelope's `EnvelopeSegment` array.

Tag Name	Description
<code>AF_TAG_ADDRESS</code>	A pointer to array of <code>EnvelopeSegment</code> points defining the shape of the envelope.

Table 5-5 Tags associated with the Envelope's `EnvelopeSegment` array.

Tag Name	Description
AF_TAG_FRAMES	The number of points in the envelope array.
AF_TAG_AUTO_FREE_DATA	Set to TRUE to cause data pointed to by AF_TAG_ADDRESS to be freed automatically when Envelope Item is deleted. The memory pointed to by AF_TAG_ADDRESS must be freeable by <code>FreeMem (Address, TRACKED_SIZE)</code> . Defaults to FALSE.
AF_TAG_TYPE	AUDIO_SIGNAL_TYPE_ constant indicating the signal type of the envelope data (i.e., the units for <code>envs_Value</code>). Defaults to AUDIO_SIGNAL_TYPE_GENERIC_SIGNED on creation.

For an example of how envelope points are described, see the example program *Examples/Audio/Misc/simple_envelope.c*. See also the Envelope Item page in the *3DO Music and Audio Programmer's Reference*.

Envelope Loops

Points define the shape of an envelope; loops control the way the envelope is played. Like a sample, an envelope can have one, two, or no loops. Possible loops are the sustain loop and the release loop. These are played just as they are in a sample. That is, the sustain loop enters after the note is started and continues until the note releases. The release loop starts only if playback enters the release stage and continues until the note is stopped. An envelope loop can be used in place of a low frequency oscillator (LFO) to produce complex modulation.

Table 5-6 Tags to control Envelope sustain and release loops.

Tag Name	Description
AF_TAG_SUSTAINBEGIN	Index in <code>EnvelopeSegment</code> array for beginning of sustain loop. -1 indicates no loop, which is the default on creation.
AF_TAG_SUSTAINEND	Index in <code>EnvelopeSegment</code> array for end of sustain loop. -1 indicates no loop, which is the default on creation.
AF_TAG_SUSTAINTIME_FP	The time in seconds used when looping from the end of the sustain loop back to the beginning. Defaults to 0.0 on creation.

Table 5-6 *Tags to control Envelope sustain and release loops.*

Tag Name	Description
AF_TAG_RELEASEBEGIN	Index in EnvelopeSegment array for beginning of release loop. -1 indicates no loop, which is the default on creation.
AF_TAG_RELEASEEND	Index in EnvelopeSegment array for end of release loop. -1 indicates no loop, which is the default on creation.
AF_TAG_RELEASETIME_FP	The time in seconds used when looping from the end of the release loop back to the beginning. Defaults to 0.0 on creation.
AF_TAG_RELEASEJUMP	Index in EnvelopeSegment array to jump to on release (ReleaseInstrument () or ReleaseAttachment () is called). When set, release causes an escape from normal envelope processing to the specified index without disturbing the current output envelope value. From there, the envelope proceeds to the next EnvelopeSegment from the current value. -1 to disable, which is the default on creation.

An envelope's sustain loop is defined by the tag arguments

AF_TAG_SUSTAINBEGIN, AF_TAG_SUSTAINEND, and AF_TAG_SUSTAINTIME_FP. The first two tag arguments set the beginning and end of the loop. Each argument is an index from the beginning point of the envelope. (The first point is 0.) If you want the sustain loop to extend from the third point of the envelope to the seventh point, then AF_TAG_SUSTAINBEGIN should equal 2 and AF_TAG_SUSTAINEND should equal 6. Note that loop beginnings and endings must fall on a point. If you want a loop point between existing points, simply add a new point to the envelope's array, then specify it with one of the sustain tag arguments.

Note that the beginning and end of a loop can be set to the same envelope point. If so, the envelope holds at that point's value as long as the loop plays.

The tag argument AF_TAG_SUSTAINTIME_FP controls what happens during playback in the return from the end of a loop to its beginning. The value in this tag argument sets the amount of time, in seconds, the envelope player takes to go from the end to the beginning of the sustain loop. The default value, 0.0, sets the envelope to return from the loop's end to its beginning without delay. Values greater than 0.0 delay the return from end to beginning.

When a loop return interval is 0.0, the envelope jumps from the end value to the beginning value. When a loop return interval is greater than 0.0, the envelope player interpolates values between the ending value and the beginning value. This creates a ramp between the end of the loop and the beginning of the loop (as shown in Figure 5-5).

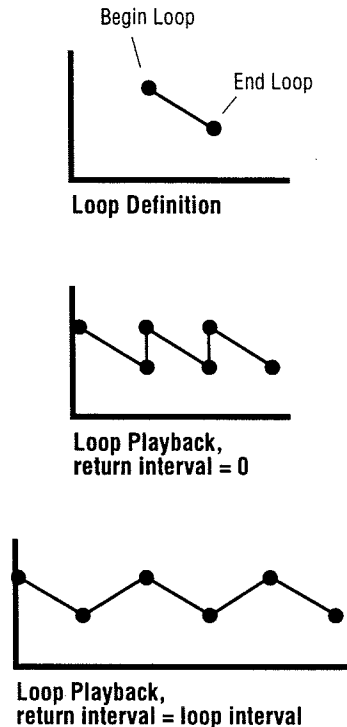


Figure 5-5 *Envelope loops.*

The release loop is also defined with three tag arguments:

AF_TAG_RELEASEBEGIN, AF_TAG_RELEASEEND, and AF_TAG_RELEASETIME_FP. These tag arguments fill the same purpose for the release loop as their counterparts do for the sustain loop.

The remaining loop control tag, AF_TAG_RELEASEJUMP, causes the envelope player to skip to a specified point immediately when the envelope is released (e.g., when `ReleaseInstrument()` is called for envelope player). When AF_TAG_RELEASEJUMP is not set, releasing an envelope has no immediate effect. Instead the envelope player continues with its normal operation, except that

when it reaches the end of the sustain loop, it continues on to the next `EnvelopeSegment`. Use of the `AF_TAG_RELEASEJUMP` tag offers much quicker response to an envelope being released.

Envelope Time Scaling

The envelope segment duration times can be scaled when the envelope is played using one or both of the following methods:

- ◆ By an arbitrary factor when the envelope player instrument is started or released.
- ◆ Based on instrument pitch when the envelope player instrument is started.

Arbitrary Envelope Time Scaling

An arbitrary time scaling factor may be applied with the `StartInstrument()` and `ReleaseInstrument()` tag `AF_TAG_TIME_SCALE_FP`. This tag accepts a floating-point time scaling factor to scale the duration times of all Envelopes attached to the Instrument which do not have the Envelope flag `AF_ENVF_LOCKTIMESCALE` set. Envelopes with `AF_ENVF_LOCKTIMESCALE` set play at their original duration times regardless of `AF_TAG_TIME_SCALE_FP`. This flag permits you to have multiple Envelopes attached to a patch, and have only some of them respond to `AF_TAG_TIME_SCALE_FP`.

The behavior of `AF_TAG_TIME_SCALE_FP` varies slightly between `StartInstrument()` and `ReleaseInstrument()`. When passed to `StartInstrument()`, `AF_TAG_TIME_SCALE_FP` scales the original duration times for all Envelopes attached to this Instrument (which do not have `AF_ENVF_LOCKTIMESCALE` set) by the specified scale factor. If not specified, all Envelopes are played with their original duration values. This setting remains in effect until the instrument is started again or is changed by `ReleaseInstrument()`. For example,

```
/* scale envelope duration times by 0.5 */
StartInstrumentVA ( ...
    AF_TAG_TIME_SCALE_FP, ConvertFP_TagData(0.5),
    .
    .
    .
    TAG_END);
```

When passed to `ReleaseInstrument()`, `AF_TAG_TIME_SCALE_FP` scales the original duration times for all Envelopes attached to this Instrument (which do not have `AF_ENVF_LOCKTIMESCALE` set) by the specified scale factor. The release scale factor replaces the start scale factor; they are not multiplied together. If not passed to `ReleaseInstrument()`, the `AF_TAG_TIME_SCALE_FP` setting passed to `StartInstrument()` remains in effect.

Pitch-Based Time Scaling

Envelopes can also be time-scaled based on instrument pitch. This is useful for imitating various acoustic instrument envelope characteristics (e.g., the amplitude envelope of a tuba is much slower than that of a trumpet; on a piano the low notes react more slowly than the high notes). The envelope tags `AF_TAG_BASENOTE` and `AF_TAG_NOTESPEROCTAVE` control how each envelope responds to the `StartInstrument()` tag `AF_TAG_PITCH`.

Table 5-7 *Tags to control Envelope time scaling.*

Tag Name	Description
<code>AF_TAG_BASENOTE</code>	MIDI note number of pitch at which pitch-based time scale factor is 1.0.
<code>AF_TAG_NOTESPEROCTAVE</code>	Number of semitones at which pitch-based time scale doubles. A positive value makes the envelope times shorter as pitch increases; a negative value makes the envelope times longer as pitch increases. Zero (the default) disables pitch-based time scaling.

For example,

```
envelope = CreateEnvelopeVA ( ...
    AF_TAG_BASENOTE,      60,      /* middle C */
    AF_TAG_NOTESPEROCTAVE, 12,
    .
    .
    .
    TAG_END);
```

In this example, the Envelope plays with its original duration times at middle C (MIDI note number 60). For each octave above middle C, the envelope duration times are halved; for each octave below middle C, the envelope duration times are doubled. Time-scaling at other intervals is exponentially interpolated (e.g., G above middle C scales the envelope duration times by approximately 2/3).

Here are some examples of this:

```
/* example envelope plays with original duration times
** at middle C */
StartInstrumentVA ( ...
    AF_TAG_PITCH, 60,          /* middle C */
    .
    .
    .
    TAG_END);

/* example envelope plays with halved duration times
** at one octave above middle C */
StartInstrumentVA ( ...
    AF_TAG_PITCH, 72,          /* octave above middle C */
    .
    .
    .
    TAG_END);

/* example envelope plays with doubled duration times
** at one octave below middle C */
StartInstrumentVA ( ...
    AF_TAG_PITCH, 48,          /* octave below middle C */
    .
    .
    .
    TAG_END);
```

Note: The envelope flag `AF_ENVF_LOCKTIMESCALE` does not prevent pitch-based time scaling.

The two time scaling systems may also be used together. When this happens, envelope duration times are scaled by both time scaling factors (i.e., `envs_Duration * AF_TAG_TIME_SCALE_FP` factor * pitch-based time scale factor). For example, using the example envelope from before,

```
/* example envelope plays with duration times scaled by 0.25 */
StartInstrumentVA ( ...
    AF_TAG_PITCH, 72,          /* octave above middle C */
    AF_TAG_TIME_SCALE_FP, ConvertFP_TagData(0.5),
    .
    .
    .
    TAG_END);
```

Other Envelope Properties

The remaining envelope tags are described in Table 5-8.

Table 5-8 *Other Envelope Tags.*

Tag Name	Description
AF_TAG_SET_FLAGS	Set of Envelope flags to set. Sets every flag for which a 1 is set in ta_Arg. The Envelope flags are described in Table 5-9.
AF_TAG_CLEAR_FLAGS	Set of Envelope flags to clear. Clears every flag for which a 1 is set in ta_Arg.

Table 5-9 *Envelope flags.*

Flag Name	Description
AF_ENVF_LOCKTIMESCALE	When set, causes the Time Scale for this envelope to ignore the use of AF_TAG_TIME_SCALE_FP in StartInstrument(). This is useful if you are using multiple envelopes in an instrument and want some to be time scaled, and some not to be. Envelopes used as complex LFOs are often not time scaled. This flag does not affect pitch-based time scaling.
AF_ENVF_FATLADYSINGS	The state of this flag indicates the default setting for the AF_ATT_FATLADYSINGS Attachment flag whenever this Envelope is attached to an Instrument.

For example, to create an envelope with time-scaling locked:

```
CreateEnvelopeVA ( ...,  
    AF_TAG_SET_FLAGS, AF_ENVF_LOCKTIMESCALE,  
    .  
    .  
    .  
    TAG_END);
```

Creating an Envelope

To define an envelope and create an envelope item, you must first create an array of `EnvelopeSegment` envelope points to define the envelope shape you want. You then use this call:

```
Item CreateEnvelope (const EnvelopeSegment *points,
                    int32 numPoints, const TagArg *tagList);
```

The call accepts three arguments: `points`, a pointer to an array of `EnvelopeSegments`; `numPoints`, a value specifying the number of elements in `points`; an optional list of envelope properties expressed as `TagArgs`.

When it executes, `CreateEnvelope()`, it creates an envelope item using the parameters you passed along with `AF_TAG_ADDRESS` and `AF_TAG_FRAMES` set to `points` and `numPoints`, respectively. If successful, it returns the item number of the envelope. If unsuccessful, it returns a negative value (an error code).

You can use `SetAudioItemInfo()` to modify many envelope parameters after creation. For example, you can use `SetAudioItemInfo()` to change an envelope's sustain loop time, pitch-based time scaling, etc.

Attaching an Envelope to an Instrument

After you have created an envelope, you must attach it to an envelope player in order to play it. Patch instruments may or may not have built-in envelope players. If an instrument has no envelope player and you wish to control it with an envelope, attach the envelope to the standard DSP instrument `envelope.dsp`, and then use `ConnectInstruments()` to connect the Output of `envelope.dsp` to the appropriate knob of the player-less instrument. To attach an envelope to an instrument, use this call:

```
Item CreateAttachment (Item master, Item slave,
                      const TagArg *tagList)
```

The call accepts three arguments: `master`, the item number of an instrument or template; `slave`, the item number of an envelope (or sample); and an optional list of `TagArgs` (described in Table 5-10).

Table 5-10 `CreateAttachment()` tag arguments for envelopes.

Tag Name	Description
<code>AF_TAG_NAME</code>	The name of the envelope hook in the instrument to attach to. <code>NULL</code> , the default, means to use the default envelope hook <code>Env</code> .

Table 5-10 *CreateAttachment () tag arguments for envelopes.*

Tag Name	Description
AF_TAG_SET_FLAGS	Set of Attachment flags (described in Table 5-11) to set. Sets every flag for which a 1 is set in ta_Arg.
AF_TAG_AUTO_DELETE_SLAVE	Set to TRUE to cause the Envelope Item to be automatically deleted when the Attachment Item is deleted. Otherwise the slave Item is left intact when the Attachment Item is deleted. Defaults to FALSE.

Table 5-11 *Attachment flags for Envelopes.*

Flag Name	Description
AF_ATTF_FATLADYSINGS	<p>If set, causes the instrument to stop when the attachment finishes playing. This flag can be used to mark the one or more Envelopes (or Samples) that are considered to be the determiners of when the instrument is done playing and should be stopped.</p> <p>The default setting for this flag comes from the Envelope's AF_ENVF_FATLADYSINGS flag setting.</p>

When `CreateAttachment ()` executes, it attaches the specified envelope to the specified envelope hook of the specified instrument or template. It creates an attachment item containing the characteristics of the attachment and returns the item number of that attachment. If unsuccessful, it returns a negative value (an error code).

The attachment flag `AF_ATTF_FATLADYSINGS` has the same effect that it has for sample attachments: when the attached envelope completes playback, the instrument to which it is attached stops.

Note that you can attach a single envelope to multiple instruments. You can also, if desired, change an envelope's data points while it is attached to an instrument. Be careful that the envelope is not played while you are changing its points: the envelope player is a high-priority task that can execute while you are in the middle of changing data.

Attaching an envelope to an Instrument Template, causes all Instruments subsequently created from that Template to also have the same Envelope attached (same Envelope Item, different Attachment Item), just as if `CreateAttachment()` had been called for each Instrument. This is useful when constructing patch templates which will be used for multiple voices.

Detaching an Envelope from an Instrument

To detach an envelope from an instrument or template, use this call:

```
Err DeleteAttachment (Item attachment)
```

It accepts a single argument: the item number of the attachment between the envelope and the instrument. When executed, it deletes the attachment item, which detaches the envelope. The call returns a non-negative value if successful, or a negative number (an error code) if unsuccessful.

Attachments are automatically deleted if either the master Item (the Instrument or Template) or Envelope Item is deleted. If the Attachment was created with `AF_TAG_AUTO_DELETE_SLAVE` set to `TRUE`, the Envelope Item is also automatically deleted when the Attachment Item is deleted. Otherwise, deleting the Attachment Item does not delete the Envelope Item.

Deleting an Envelope

If an envelope is no longer in use, you should delete it to free system and task resources. To do so, use this call:

```
Err DeleteEnvelope (Item envelope)
```

The call accepts the item number of the envelope to be deleted. When it executes, the call deletes the envelope item and any attachments to it. If successful, it returns a non-negative value. If unsuccessful, it returns a negative number (an error code).

By default, when an Envelope is deleted only the Envelope Item and its Attachment Items are deleted, not the `EnvelopeSegment` array pointed to by the Envelope Item. However, if the Envelope was created with `AF_TAG_AUTO_FREE_DATA` set to `TRUE`, then the `EnvelopeSegment` array is automatically freed when the Envelope Item is deleted. To avoid referencing freed memory, it is important to make sure that nothing else points to such an `EnvelopeSegment` array.

Advanced Audio Folio Usage

This chapter describes audio folio techniques. It contains the following topics:

Topic	Page Number
Introduction	87
Reading and Changing Audio Item Attributes	87
Reading Instrument Output	89
Tuning Instruments	90
Adding Reverberation	94
A Delay Line Overview	94
Audio Clocks	99

Introduction

Most features of Advanced Audio Folio are not required to write simple applications. The information about how to change audio item attributes, tune instruments, and use probes is included for developers who wish to maximize their use of the 3DO Audio System.

Reading and Changing Audio Item Attributes

Many Audio folio items have attributes that must be changed after the item has been created. For example, a sample created for use with a multisample instrument can be changed so that its pitch or velocity range can be limited. Because items are system resources, a task cannot alter them directly. To provide a task with the ability to change audio items, the Audio folio provides one call that

reads an item's current settings and a second call that changes an item's current settings. These calls work with sample, envelope, attachment, knob, and tuning items.

Reading Audio Item Characteristics

To read the current tag argument values for a sample, envelope, attachment, tuning, or knob item, use this call:

```
Err GetAudioItemInfo(Item audioItem, TagArg *tagList)
```

The call accepts the item number of the item for which you want information, and a pointer to a list of tag arguments. You must set up the tag argument list before you use the call. You should include any of the item's tag arguments in which you are interested, anywhere from only one tag argument to all of the appropriate tag arguments. (The tag arguments are listed in the description of `SetAudioItemInfo()` in Chapter 1, "Audio Folio Calls," of the *3DO Music and Audio Programmer's Reference*.

When `GetAudioItemInfo()` executes, it finds the current value for each of the tag arguments in the list and writes that value to the tag argument. The call returns 0 if successful, or a negative number (an error code) if unsuccessful.

To get the information you asked for, read the values written into your tag argument list. (To make sure that each value is written, it is wise to initially set the tag argument values to NULL, or some other unlikely value. `GetAudioItemInfo()` writes over the original values.)

Setting Audio Item Characteristics

To change the characteristics of a sample, envelope, attachment, or tuning item, (but not a knob item), use this call:

```
Err SetAudioItemInfo(Item audioItem, const TagArg *tagList)
```

The call accepts the item number of the item you want to change. It also accepts a pointer to a list of tag arguments that you must set up before you use the call. The tag argument values you supply in the list determine the characteristics of the item. (The tag arguments are described in Chapter 1, "Audio Folio Calls," of the *3DO Audio and Music Programmer's Reference* under the description of `SetAudioItemInfo()`.) You can also use `SetAudioItemInfoVA()`. See Chapter 1 of the *3DO Audio and Music Programmer's Reference* for more information.

When `SetAudioItemInfo()` executes, it reads the tag argument values in the tag argument list and applies them to the specified item. It returns 0 if successful, or a negative value (an error code) if unsuccessful.

Note that this call does *not* work with any audio items other than sample, envelope, attachment, or tuning items.

Reading Instrument Output

You can directly read the output of an instrument by creating a Probe item for the output and then using it to read the output of the instrument. Creating the Probe is analogous to creating a Knob item for an instrument except that the purpose of the Probe is to read and the purpose of the Knob is to write.

Creating the Probe

Before a task can use a Probe to read instrument output, it must first create that Probe by calling the `CreateProbe()` function:

```
Item CreateProbe( Item Instrument, const char *portName,
                  const TagArg *tagList )
```

The `CreateProbe()` function takes three arguments: the item number of the instrument, a pointer to a string containing the name of the output the task wants to probe, and a tag argument. If the `CreateProbe()` call is successful, the call creates a probe item connecting the probe and the task, and returns its item number. If unsuccessful, it returns a negative value (an error code).

Using Probes

If you are reading a single-part Probe, use the `ReadProbe()` call:

```
Err ReadProbe (Item probe, float *valuePtr)
```

The call accepts two arguments: `probe`, the Item number of the knob to be tweaked; and `ValuePtr`, a pointer to where to put result. The task must have write permission for the address specified by `ValuePtr`.

`ReadProbe()` reads the instantaneous value of an instrument output. It is useful for reading the output of slowly changing instruments such as `envelope.dsp`, `envfollower.dsp`, `minimum.dsp`, and `maximum.dsp`. It replaces the now obsolete function `DSPReadEO()`.

`ReadProbe()` returns a non-negative value if successful, or an error code (a negative value) if unsuccessful.

If you are probing a multi-part output, use this call:

```
Err ReadProbePart (Item probe, int32 partNum, float *valuePtr)
```

This call accepts three arguments. Two of them, `probe` and `valuePtr`, are identical to those used with `ReadProbe()`. The third argument, `partNum`, is the index of the part to be read. `ReadProbePart()` functions identically to `ReadProbe()`. It returns a non-negative value if successful, or an error code (a negative value) if unsuccessful.

Tuning Instruments

Once you have created instruments and started to play them, you can specify given pitches for instrument notes. A pitch is a discrete value that corresponds to a given frequency—you might want to think of it as an index number in a frequency lookup table. The most common example of specifying pitch is by name—middle C, for example, or G an octave and a fifth above that. You also specify pitch whenever you press a key on a standard keyboard instrument, the fifth black key from the bottom, or the white key at the top of the keyboard. In both cases, you do not worry about the frequency of the note; the instrument you are using translates the specified pitch into the frequency you want.

MIDI instruments, for convention's sake, specify pitch as an integer from 0 to 127. They tie those values to the pitches used in a standard 12-tone, equal-tempered tuning system (the kind you find on almost every standard western keyboard instrument). The pitch value 60 corresponds to middle C and each count up or down moves up or down one half-step in pitch. For example, 61 is a C sharp (or D flat), 62 is a D, 59 is a B, 58 is a B flat (or A sharp), and so on. A jump of 12 is an octave. For example, 69 is an A 440; 81 is an A 880, one octave above A 440.

Portfolio instruments use the standard MIDI tuning system by default, built on pitch values from 0 to 127, playing a 12-tone, equal-tempered tuning system. Not everything sounds its best in this tuning. If you want to reproduce the sounds of a Javanese gamelan or an Indian raga, the out-of-tune splendor of a honky-tonk piano, or the tight-knit harmonies of a barbershop quartet, you have to come up with some alternative tunings that assign different frequencies to the 128 MIDI pitch values.

You can, of course, specify the precise frequency as you play each note to get just the tuning you want. But if you want to play a MIDI score directly on an instrument without setting the precise frequency for each note, that will not work. You can, instead, reset the tuning of the instrument before you play. The score then plays back using the appropriate tuning system. To help you create and use alternative tunings, the Audio folio offers a set of tuning calls.

Creating a Tuning

(A note to those who are happy with standard MIDI tuning: you can skip this part of the tuning section and use instruments with their default tunings. You might be interested, however, in the final tuning topic here, bending pitch.)

To create a tuning, you must create a list of frequencies (measured in cycles per second or Hertz) that you want to associate with MIDI pitch values. Each frequency value is stored as a float32. The list of frequencies normally goes from lowest to highest, although you can invert that or mix the frequencies to create pitch inversion or true havoc when playing back MIDI scores.

You can, of course, create a list of 128 different frequencies, one for each of the 128 MIDI pitch values. This is useful if your tuning is irregular, and does not repeat itself from one octave to the next. If, however, your tuning uses one set of frequencies within an octave, and that set repeats in higher and lower octaves, then you need only define the frequencies within that one octave. For example, if you want to create a quarter-tone tuning system with 24 pitches per octave, you need only define the 24 pitches within a single octave. The Audio folio extends the tuning to the other octaves. You can also define two or more octaves of pitches if the tuning changes from one octave to the next, but repeats every third, fourth, or fifth octave.

Before you can apply a list of frequencies to an instrument, you must create a tuning based on that frequency list by using this call:

```
Item CreateTuning(const float32 *Frequencies, int32 NumNotes,
                  int32 NotesPerOctave, int32 BaseNote)
```

The first value this call accepts is a pointer (*Frequencies) to the list of frequencies. The second value (NumNotes) is the number of frequencies found in the list. The third value (NotesPerOctave) contains the number of pitches that fall within a single octave. And the final value (BaseNote) is a single MIDI pitch value that corresponds to the first frequency in the frequency list.

When CreateTuning() executes, it reads the frequencies in the list, starting at the beginning and going the number of frequencies specified by NumNotes. It is important to set NumNotes correctly to avoid losing frequencies or reading garbage values. It assigns the first frequency to the value set by BaseNote and assigns subsequent frequencies to subsequent values. For example, if a five-frequency list is assigned to pitches starting with a base pitch of 60, the five frequencies are assigned in order to 60, 61, 62, 63, and 64. CreateTuning() returns the item number of the tuning created if successful, or a negative value (an error code) if unsuccessful.

Extending Tuning

If a tuning does not contain 128 independent frequencies, then many pitch values do not have an assigned frequency. This is where the NotesPerOctave value comes into play. If an instrument plays a note that falls outside the assigned range of pitches, the Audio folio looks at the number of pitches contained in each octave and extends the defined pitches to assign an appropriate frequency to undefined pitches.

For example, consider the five-frequency list used above. Assume there are five pitches per octave. If an instrument using the tuning is asked to play pitch value 65 (which is unassigned), the Audio folio looks down five pitches (the span of an octave), finds the frequency assigned to 60, and doubles it to produce an octave-higher frequency for 65. To play pitch value 70, the Audio folio drops down 10 pitches (the span of two octaves) to 65 and quadruples the frequency there to produce a frequency two octaves higher.

If you are working with a sampled sound instrument, it typically has a pitch range that extends no higher than one octave above its recorded pitch. Below its recorded pitch, its pitch range descends into the depths of inaudibility.

Applying a Tuning

Once you have created a tuning, you can use it by applying it either to an instrument, which affects notes played only by that instrument, or to an instrument template, which affects notes played by all instruments created using that template.

To apply a tuning to one instrument, use this call:

```
Err TuneInstrument( Item Instrument, Item Tuning )
```

The call accepts the item number of the instrument and the item number of the tuning you want to apply to the instrument. When it executes, it applies the tuning to the instrument so the instrument plays with that tuning. The call returns 0 if successful, or a negative value (an error code) if unsuccessful.

To apply a tuning to an instrument template (thereby applying it to all instruments created using that template), use this call:

```
Err TuneInsTemplate( Item InsTemplate, Item Tuning )
```

The call accepts the item number of the instrument template and the item number of the tuning you want to apply to the template. When it executes, it applies the tuning to the instrument template so that all instruments associated with the template play using that tuning. The call returns 0 if successful, and a negative value (an error code) if unsuccessful.

After tuning an instrument template, all instruments created from the template use the tuned template.

Deleting a Tuning

When a task finishes using a tuning, it should delete it to free system resources. To do so, use this call:

```
Err DeleteTuning( Item Tuning )
```

The call accepts the item number of the tuning. When executed, it deletes the tuning. It returns 0 if successful, or a negative value (an error code) if unsuccessful.

If an instrument is using a tuning and the tuning is deleted, the instrument returns to its default tuning.

Bending Pitch

Most standard MIDI synthesizers contain a device called a pitch wheel. When you move the pitch wheel up or down, it bends the overall pitch of the synthesizer up or down. The current setting of the wheel determines how much the overall pitch of the synthesizer bends up or down.

The Audio folio provides a similar mechanism for audio instruments. Each instrument stores a bend value (a float32 value) that multiplies the instrument's output frequency, bending any pitches played up or down by an amount corresponding to the multiplier. For example, if an instrument has a bend value of 2, then all output frequencies are doubled, so the instrument's output pitches bend up by an octave in pitch. If an instrument has a bend value of 0.5, then all output frequencies are halved, so the instrument's pitches bend down by an octave in pitch. And if an instrument has a bend value of 1 (the default value), the output frequencies do not change, and the instrument's output pitches are not bent.

To set an instrument's bend value, use this call:

```
Err BendInstrumentPitch( Item Instrument, float32  
    BendFrac )
```

The call accepts two arguments: *Instrument*, the item number of an instrument; and *BendFrac*, a bend value to apply to that instrument. When it executes, the call applies the bend value to the instrument. All notes played on that instrument are then bent by the amount specified. The call returns 0 if successful, or a negative value (an error code) if unsuccessful.

An instrument's bend value affects all pitches equally, so it does not affect the instrument's relative tuning. For example, if you set an instrument's tuning to a 24-step quarter-tone scale and then you bend the instrument up by a minor third, the instrument still plays a 24-step quarter-tone scale. It simply plays all the pitches in the scale a minor third higher than specified in the tuning.

Creating a Bend Value

Because most musically astute humans tend to think of pitch intervals as a number of half-steps fine-tuned up or down by cents (100 cents equal a half-step), determining a floating-point bend value may not be a simple matter. To create a float32 bend value easily, call the `Convert12TET_FP()` function:

```
Err Convert12TET_FP( int32 Semitones, int32 Cents,  
                    float32 *FractionPtr )
```

The `Convert12TET_FP()` function takes three arguments: `Semitones`, an integer number of half-steps (semitones) to bend up or down; `Cents`, an integer number of cents to bend up or down; and `*FractionPtr`, a pointer to a floating-point variable where the created bend value is stored. When `Convert12TET_FP()` executes, it combines the number of half-steps and the number of cents to determine the final bend interval, creates a bend value that matches that interval, and then writes the bend value to the variable provided. If successful, it returns 0; if unsuccessful, it returns a negative value (an error code).

The `Semitones` and `Cents` arguments used for `Convert12TET_FP()` can be negative or positive. Negative values bend pitch down, positive values bend pitch up. The two arguments need not both be positive or both be negative, one can be positive while the other is negative. For example, -5 semitones and +30 cents specifies a downward pitch interval of 4 semitones and 70 cents (an interval that is a little less than a fourth).

Adding Reverberation

Reverberation is an effect that adds reality to a sound and gives a listener acoustic cues about the size and reverberant qualities of the sound's surroundings. To add reverberation to an audio signal, the Audio folio offers two reverberation tools that must be used together: a *delay line*, which stores an audio signal; and a *delay instrument*, which writes an audio signal into a delay line. These are combined with other instruments to create simple delay effects such as a single echo or complex delay effects such as rich reverberation.

The simplest way to add reverberation to an application is to use one of the Reverberation patches described in Chapter 7, "Patch Templates". The patches are supplied as source code which you can modify to suite your application. If you would like to create delay effects, without using patches, or you want more information about how to create delays, continue with the material about Delay Lines.

A Delay Line Overview

A delay line is, in essence, a special sample that has its sample data stored in system memory (unlike a standard sample, which has its sample data stored in task memory). A delay line exists to accept samples written to it by a delay instrument.

A delay instrument (such as `delay_f1.dsp`) accepts an audio signal input from another source such as a sampled sound instrument or a sound synthesis instrument. The delay instrument writes that audio signal into the delay line, starting at the beginning of the delay line. It writes samples through to the end of

the delay line, then starts over at the beginning, overwriting previously written samples. This looping recording of audio signal samples provides a stored record of the audio signal. Its duration depends on the size of the delay line. For example, a 44,100 sample delay line can store one second's worth of audio signal.

To create a delayed playback of the audio signal stored in the delay line, a sampled sound instrument is attached to the delay line. The instrument reads the delay line sample as it would a normal sample, with two differences: the instrument reads the sample and loops back to read it again and again; and the attachment from the delay line to the instrument is set to start sample playback at a point later than the starting point. (To do this, use the attachment's `AF_TAG_START_AT` tag argument.) The result is that the sampled sound instrument's playback trails the delay instrument's sample writing as shown in Figure 6-1.

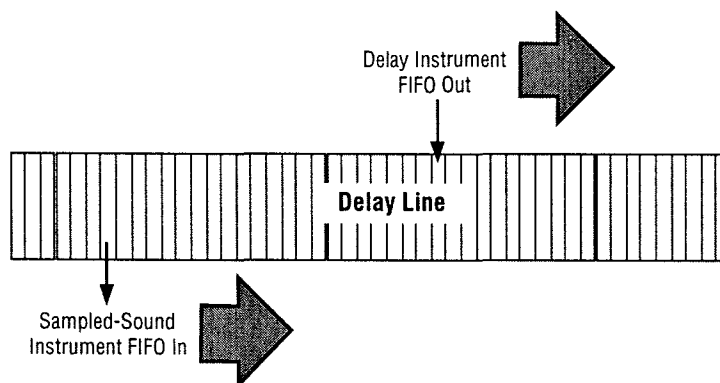


Figure 6-1 Sample of a delayed audio signal.

The number of samples between the delay instrument's writing and the sampled sound instrument's reading sets the amount of time the audio signal is delayed. For example, the delay instrument writes to the delay line at 44,100 samples per second. The sampled sound instrument reads at the same rate (you should never use a 22,050 Hz sampled-sound instrument to read the delay line), but is 11,025 samples behind. The total delay is 1/4 second (11,025 divided by 44,100). For an example, see *Examples/Audio/Misc/ta_customodedelay.c*.

Creating a Delay Line

To create a delay line, use this call:

```
Item CreateDelayLine (int32 numBytes, int32 numChannels,
                     bool ifLoop)
```

The call accepts three arguments: `numBytes`, the size of the delay line's sample buffer in bytes; `NumChannels`, the number of channels stored in each sample frame; and `IfLoop`, a setting that determines whether the delay line is written to repeatedly or only once.

The first value, `numBytes`, should be determined by the length of the audio signal you want stored, by the size of each sample, and by the number of channels in each frame. For example, if you want to store two seconds of a 16-bit stereo signal, you should set `numBytes` to 352,800 bytes (2 seconds times 44,100 Hz times 2 bytes times 2 channels).

The second value, `numChannels`, is usually 1 for a mono sample and 2 for a stereo sample. Note that the Audio folio does not handle more than two channels in a sample.

The third value, `ifLoop`, is either `TRUE` (1) to set the delay line so that the delay instrument writes to it continuously, looping sample data; or `FALSE` (0) to set the delay line so that the delay instrument writes to the delay line only once and then stops.

When it executes, `CreateDelayLine()` allocates a sample data buffer in system memory, creates a delay line item, and returns the item number of the delay line if successful. If unsuccessful, it returns a negative value (an error code).

Note that the task cannot write directly to the sample data buffer, because it is located in system memory. If the task attempts to write to system memory, it will crash. The task must use the delay instrument to write into the delay line's sample data buffer.

Deleting a Delay Line

When a task finishes using a delay line, it should delete it to free system resources. To do so, use this call:

```
Err DeleteDelayLine( Item DelayLine )
```

The call accepts the item number of the delay line to be deleted. When executed, it deletes the delay line item. It returns 0 if successful, or a negative value (an error code) if unsuccessful.

Connecting a Delay Line to Create Reverberation

Once you create a delay line, you must connect it to a suitable configuration of instruments to create reverb effects. Figure 6-2 shows a typical and fairly simple set of connected instruments and attached items. See the example file *ta_customdelay.c* for a program example. The principle elements are:

- ◆ A source-audio signal, typically from a sampled sound instrument, sound synthesis instrument, or a submixer. (Figure 6-2 shows a sampled sound instrument.)
- ◆ A submixer to feed the source audio signal to the reverb loop, and to mix the resultant reverb signal with the source signal and back into the reverb loop for more complex reverb.
- ◆ A delay instrument to read the source audio signal and write it into the delay line.
- ◆ A sampled sound instrument to read the delay line (starting behind the delay instrument, of course) and to feed the result back to the submixer.

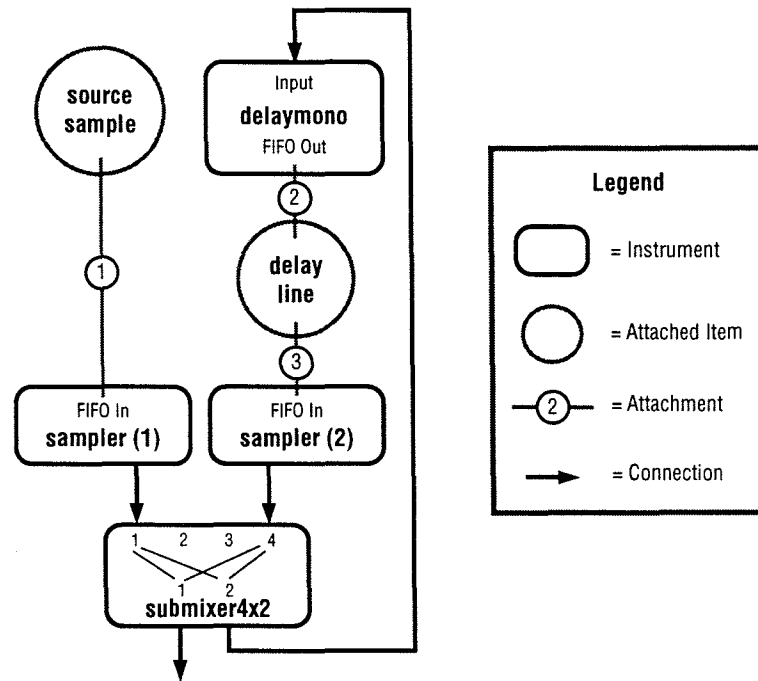


Figure 6-2 A typical reverberation setup using a delay line and a delay instrument.

Setting the Attachment Starting Point

One very important setting in the reverb configuration shown in Figure 6-2 is the starting point of attachment 3, which attaches the delay line to the sampler that reads it. You must use the attachment's `AF_TAG_START_AT` tag argument to specify a point in the delay line to start playback. The amount of delay is determined by the distance of the starting point from the end of the delay line. If sample playback starts at the last sample in the delay line, then it is only one

sample behind the writing, a delay of only 1/44,100 of a second. If sample playback starts 11,025 samples from the end of the delay line, then the delay is 1/4 of a second.

Caution: `AF_TAG_JTARIAT` is specified in frames while the length of the delay is specified in *bytes* using `CreateDelayLine()`. There is a potential for confusion here.

As a practical matter, the sample playback should never be within 32 samples of the writing point, because the DSP writes to the delay line in bursts of samples. If the reading and writing instruments are too close in the delay line, you can get some inconsistencies in the delayed audio signal. To get maximum delay time, sample playback should start at the twentieth sample of the delay line, which gives as much delay as the length of the delay line will allow and still provide for a 32-sample read/write separation.

Another consideration when planning the distance of the playback point from the writing point is that when you start the two instruments with two `StartInstrument()` calls, a task of higher priority may briefly steal cycles between the two calls and throw off the timing. You should first start the instrument that is a greater circular distance behind the other instrument, then start the second instrument. This makes it less likely that the two instruments will be so close together that no delay results.

It is possible to combine the playback and writing instruments into one with `MakePatch`. You can then be guaranteed that you start both instruments simultaneously.

Setting the Submixer Levels

Figure 6-2 shows the source signal entering input 1 of a 4 x 2 submixer and the reverb signal entering input 4. It also shows two outputs: Output 1 of the submixer goes off either to a mixer, where it is sent to the DAC for playback or to another instrument for further processing. Output 2 feeds back to the reverb loop. The balance of the mixer determines how the reverb will sound.

For a simple echo, output 1 is set to put out the source signal mixed with a diminished reverb signal. Output 2 is set to put out only the source signal. Because the reverb loop receives only the source signal, it creates a single echo (a delayed playback) of the source.

For many receding echoes, output 1 is set, as before, to put out the source signal mixed with a diminished reverb signal. Output 2 is also set to put out the source signal mixed with a diminished reverb signal. Because the reverb signal feeds back into the reverb loop, you create a series of receding echoes that you can adjust with the mix of the reverb to source signal and with the length of the delay set by the attachment starting point.

Adding Extra Delay Line Readers

For truly complex reverberation, where the echoes are not regularly spaced, you can attach several sampled sound instruments to the delay line and feed their inputs to the submixer. Each sample sound instrument plays back the delay line starting at a different point and their outputs are mixed back into the reverb loop. To avoid an audible periodicity in the echoes, you should make sure that the starting points of the different sampled-sound instruments are irregularly spaced in the delay line.

Using a Delay Line for Oscilloscope Data

A delay line is a useful tool for more than adding reverb to an instrument. You can also use it to capture a snippet of an audio signal and then use it for oscilloscope display. To do so, simply create the delay line so that it does not loop. Feed an audio signal into it using a delay instrument, and when the delay line is full, read it for display.

Audio Clocks

The Audio Folio supports multiple audio clocks which can be used for scheduling musical notes or other events.

The functions `GetAudioTime()` and `SleepUntilTime()` use a single global clock that is shared by all tasks. This global clock is generated by dividing the 44100 Hertz sample rate clock by 184. The value 184 is the *duration* of the clock in sample frames. Dividing the sample rate by the clock duration results in a clock rate of approximately 239.674 Hertz. This clock rate is close to the 240 Hertz clock that is common in SMPTE and other musical timing applications.

The clock rate of the global clock cannot be changed because it would impact other tasks. If you need a clock that runs at a different clock rate for a task, then you can create a custom `AudioClock` for that task. The rate of that clock may be changed freely.

Creating a Custom Audio Clock

To create a custom `Audio Clock`, use the Audio Folio function call `CreateAudioClock()`.

```
Item CreateAudioClock( const TagArg *tagList );
```

You can pass `NULL` for the `tagList`. By default, the clock will have the same duration as the global clock.

Setting the Clock Rate

There are two ways to set the rate of your custom clock. The most direct and precise way is to specify a new duration using `SetAudioClockDuration()`. You can call `SetAudioClockRate()` which takes a floating point value in Hertz.

```
Err SetAudioClockDuration( Item clock, int32 numFrames );  
Err SetAudioClockRate( Item clock, float32 hertz );
```

`SetAudioClockRate()` will calculate a duration that yields a rate closest to the one you have specified. The actual rate will probably not be exactly what you specified since it is a calculated value. To find out the actual rate, you can call the query function `GetAudioClockRate()`.

```
Err GetAudioClockRate( Item clock, float32 &hertz );
```

You can query the rate of the shared global clock by passing `AF_GLOBAL_CLOCK` instead of the item number.

If you are trying to synchronize precisely with an audio data stream, it would be best to use `SetAudioClockDuration()` and to do all of your calculations in sample frames. The reason for not using the clock rate is that the value is inherently imprecise and using it may cause your application to drift slightly in relation to an output audio stream.

Function Calls

The following Audio folio calls provide advanced instrument control.

Tuning

The following calls can be used to tune items and instruments:

<code>CreateTuning()</code>	Creates a tuning item.
<code>DeleteTuning()</code>	Deletes a tuning.
<code>TuneInsTemplate()</code>	Applies the specified tuning to an instrument template.
<code>TuneInstrument()</code>	Applies the specified tuning to an instrument.
<code>Convert12TET_FP()</code>	Converts a pitch bend value, in semitones and cents, into a float value

Adding Reverberation

The following calls are used to create and delete delay lines:

<code>CreateDelayLine()</code>	Creates a delay for echoes and reverberations.
<code>DeleteDelayLine()</code>	Deletes a delay line.

Audio Clocks

The following calls use a global clock, create a custom clock, set the rate of a custom clock, and obtain the rate of a custom clock.:

<code>GetAudioTime()</code>	Returns the audio time from the global clock.
<code>SleepUntilTime()</code>	Puts calling task to sleep until the global clock reaches the specified audio time.
<code>CreateAudioClock</code>	Creates a custom Audio Clock, different from the Global clock.
<code>SetAudioClockDuration()</code>	Specifies the duration of the Audio Clock's tick.
<code>SetAudioClockRate()</code>	Query the rate of the Audio Clock.
<code>GetAudioClockRate()</code>	Query to the Global clock.
<code>SignalAtTime()</code>	Request a signal be sent to the calling task when the global clock reaches the specified audio time.
<code>ReadAudioClock()</code>	Query the current time of an audio clock.

Patch Templates

This chapter describes how to prepare Patch Templates for use with the Audio Folio. It contains the following topics:

Topic	Page Number
Introduction	103
Patch compiler	104
Binary Patch File Loader	104
Makepatch	104
Patch Examples	104
Reverberation Examples	110

Introduction

A patch template is a custom instrument template created from constituent instrument templates with defined internal connections, custom knobs and ports. They may be used anywhere any other kind of instrument template may be used, including attaching samples and envelopes to them, creating instruments from them, using them in a MIDI score, constructing more complicated patches, etc.

Patch template support in the 3DO M2 Portfolio consists of the following components:

- ◆ Patch compiler, which resides in the audiopatch folio.
- ◆ Binary patch file loader, which is in the audiopatchfile folio.
- ◆ `makepatch`, which is a 3DO shell program to create binary patch files.

Patch Compiler

The patch compiler is a run-time service of the audiopatch folio, that constructs a custom patch template. The language used to describe the patch to the patch compiler is a PatchCmd list, which is conceptually similar to a TagArg list. This is the lowest level of patch support.

For more information, see the Audio Patch Folio Calls in Chapter 1 of the *3DO Audio and Music Programmer's Reference*. See `windpatch.c`, in the *3DO Supplementary Portfolio Reference* for an example of direct use of the patch compiler.

Binary Patch File Loader

A binary patch file is an IFF file which stores a representation of a PatchCmd list, along with any samples or envelopes to attach to the patch. The audiopatchfile folio provides a loader for binary patch files. The loader uses the patch compiler to build the actual patch template, and then attaches all of the samples and envelopes, stored in the patch file, to the patch.

See the Audio Patch File Folio Calls, in Chapter 2 of the *3DO Audio and Music Programmer's Reference* for more information.

Makepatch

makepatch is a 3DO shell program for making binary patch files, using a simple patch script language.

See the *3DO Supplementary Portfolio Reference* for more information about the makepatch shell command, and an example.

Patch Examples

The 3DO M2 *Portfolio and Toolkit Release 2.0* CD includes a set of sample patch definitions in the directory *Examples/Audio/Patches*. They are intended to show you how to use the patch language interpreted by makepatch, as well as provide working sounds that you can customize and include in your own programs.

Here's how you use these patches:

- ◆ If you \$samples alias has not already been set by the DevStartup script, then tell the system where to find your samples directory using the alias shell command (for example, `alias samples /remote/samples`).
- ◆ Run the script `makepatch.script`, which uses the shell program makepatch to generate binary files for each of the examples.
- ◆ Run each of the resulting binary patch files using the shell program `dspfaders` (or any other program which loads binary patch files).

In building these patches, we follow a few simple conventions:

- ◆ Where applicable, patches include the knobs Amplitude and either Frequency or SampleRate. This allows them to be controlled by the Audio Folio's MIDI player software so that, for example, velocity and note number can affect the sound of the patch. We use SampleRate when the knob controls the rate of playback of one of the sampler instruments, and Frequency otherwise. Note that these knobs don't necessarily control only amplitude and frequency values; Amplitude may also control the cutoff frequency of a filter or the decay time of a reverberator, for example.
- ◆ Patch output runs through a single- or multi-part output called, appropriately, Output. Using this name ensures that dspfaders connects the output to the hardware's audio out so you can hear the result. Similarly, patches that take audio input have a single- or multi-part input called Input.
- ◆ Patch commands tend to be grouped logically: first, instruments are declared, then audio connections, then knobs and constants, then control connections.
- ◆ Knob names indicate both what the knob controls, and how (albeit somewhat cryptically at times, given space constraints). For example, a knob called FreqModDepth is preferable to one called Depth.
- ◆ Where possible, the -FatLadySings flag is used with envelopes, thereby stopping the instrument when the envelope runs out. This leads to better voice allocation when playing patches in multi-voice applications.
- ◆ When we line up columns of text, we use spaces rather than tabs. This way, regardless of tab width, columns will still line up.

The following sections describe each of the example patches found in the directory */remote/Examples/Audio/Patches* on the *3DO M2 Portfolio and Toolkit Release 2.0 CD*.

sample.mp

One of the first things you'll want to try is playing back samples from the sample library accompanying the system. Samples need a sample-playing instrument to attach to, so we use a `sampler_16_v1.dsp` instrument, which plays one channel of 16-bit data with a variable sample rate. See "Loading and Attaching Samples" in Chapter 4, for more information on how to attach a sample to a sample-playing instrument.

Note that the patch uses the aliased pathname `$samples`, which must be set before compiling the patch. See the "Shell Commands" chapter, in the *Supplementary Portfolio Reference* for more information.

Note that we include knobs for varying the amplitude and sample rate, so that we can play the sample at different pitches and loudnesses. This allows the patch to be used as a MIDI instrument with programs such as `playmf`.

lfo_freq_tri.mp

This is a purely synthetic patch. The sound is produced by a pulse wave oscillator whose frequency is controlled by a slowly-changing triangle wave oscillator. We use a `triangle_lfo.dsp` instrument ("lfo" = low frequency oscillator) for the latter, which allows fine control of low frequencies at the expense of not being able to run at frequencies above about 86 Hz.

The output of the lfo is scaled and shifted using a `timesplus.dsp` instrument so that it varies within a range (defined by the multiplication factor *InputB*) around a base value (defined by the addition factor *InputC*), providing vibrato. This is a generally useful technique for controlling a modulation source. The range is connected to a knob, *ModDepth*. The vibrato rate is controlled by the frequency of the lfo, and adjusted by a knob called *ModRate*.

As usual, we include knobs for frequency and amplitude. The frequency of the patch is really defined by the base frequency of the vibrato, which is controlled by the value in *InputC* of the `timesplus.dsp` instrument, so that's what the Frequency knob is connected to.

lfo_amp_pulse.mp

Here's another example of modulation. This time, the modulating oscillator is a pulse wave, a variable-width square wave. The scaled and shifted output controls the amplitude of the oscillator you hear, a regular square wave. With the modulation scaling/shifting knobs (*ModDepth* and *Amplitude*) at the default values, the sound effectively changes from full volume to zero volume and back again at a rate determined by the *ModRate* knob. Note that you can make the amplitude of the square wave negative by setting *ModDepth* higher than *Amplitude*; when the square wave oscillator is given a negative amplitude, you effectively reverse the phase of the output sound. You can also overdrive the audio signal, by setting *ModDepth* and *Amplitude* so that they add to more than 1.0 or less than -1.0.

tri_amp_samp.mp

You can use a modulator with a sample instrument, too. This patch uses a similar setup to that of `lfo_freq_tri.mp`, this time with the lfo modulating the amplitude of the sample, providing tremolo.

rhold_freq.mp

There are other instruments that make nice control sources, apart from the LFOs. This patch uses the `randomhold.dsp` instrument to randomly change the pitch of a `sawtooth.dsp` instrument at a rate determined by the *ModRate* knob. Again, a `timesplus.dsp` is used to scale and shift the pitch variations.

saw_pulse_svf.mp

With this example of modulation, there are two independent modulators controlling two independent parameters. This is a good example of the difference in accuracy at low frequencies between an audio-rate dsp oscillator and an lfo-rate one. When you run this patch with dspfaders, you'll notice phasing between the filter and pitch sweeps (both set in the source below to 3.0 Hz). You won't be able to adjust the `FiltRate` slider to compensate exactly, because the filter sweep uses the `sawtooth.dsp` instrument, whose `Frequency` knob doesn't have sufficiently fine precision to be hit 3.0 Hz exactly. The precision is approximately 0.673 Hz; the nearest multiples of this to 3.0 are 2.692 Hz and 3.365 Hz. An LFO's frequency precision is 256 times as fine as that of an audio frequency oscillator (approx. 0.00263 Hz), so `triangle_lfo.dsp`'s `Frequency` knob can be set almost exactly to 3.0 Hz (either 2.999 Hz or 3.002 Hz).

You can get all sorts of interesting things out of this patch by setting the `Frequency` knob to a small negative value (for example, -4.0).

helicopter.mp

This patch creates a fairly reasonable approximation of a helicopter's rotor using the modulation concepts introduced in prior patches. Unlike a sample of that sound, you can vary the sound significantly by tweaking the knob values, making the helicopter sound closer or more distant, airborne or on the ground, and so on. If you were to do this with samples, you'd need different samples for each situation, which quickly uses up available memory. Furthermore, it wouldn't be possible to continuously vary the sound from one form to another as it is with this synthetic patch.

These sorts of considerations are generally applicable to using sound on the 3DO M2. Spending the time to come up with a patch that is totally synthetic, or uses some combination of samples and generating/processing instruments, generally results in a much more flexible instrument that takes less memory than a set of samples.

Note the use of the `svfilter.dsp` instrument. This provides low-pass, band-pass and high-pass filtered output simultaneously, and is controlled by *Frequency* and *Resonance* parameters. If the resonance parameter is set low enough, the filter will oscillate. The frequency parameter is unfortunately not the same as oscillator frequencies, but is related in a non-linear way. At low frequencies (less than about 8KHz), the match is close enough that it generally doesn't matter, unless you're driving the filter into oscillation and trying to exactly tune the resulting sound (so you can play it as a MIDI instrument, for example).

rocket.mp

Here's an example of a patch that sounds like a rocket engine. The basic sound is provided by a `rednoise.dsp` instrument, whose frequency and amplitude are

themselves modulated by other `rednoise.dsp` instruments. The values chosen for shifting/scaling the frequency and amplitude controls create a rumbling sound with a few high-frequency peaks. By connecting knobs to these parameters, as well as the frequency of the modulating rednoise instruments, you can change the character of the rocket (inside, outside, close, far, and so on). Again, the variability of the patch makes it potentially much more useful than a simple sample or set of samples.

wind.mp

In this example, the sound of wind is created from a patch using modulation. Rather than looping a wind sample, you can make a continuously varying wind sound using filtered white noise with random modulation. The whistling sound comes from driving the filter into oscillation with low resonance values.

snare_delay.mp

This is a simple example of using a delay line. The input from the snare sample instrument runs into a mixer, which has two outputs: one goes directly to the audio output, and the other into the delay. The output of the delay runs into the other channel of the mixer, which sends some to the audio output and some back into the delay. This means raw input circles around through the delay/mixer; if the gain on the loop is less than 1.0, the sound will decay away to silence. By setting the gain on the feedback loop negative, the whole patch becomes more stable since any DC components in the signal cancel out very quickly.

As with samples, delay lines are static; they need to be attached to corresponding instruments to be used. Unlike samples, delay lines are attached to more than one instrument; they usually have a writer instrument (here, it's a `delay_f1.dsp` instrument) that passes samples in to the line, and one or more reader instruments (here, a `sampler_16_f1.dsp` instrument) to pull delayed samples back out.

The total delay time possible for a given delay line depends on its length, and the rate of the reader and writer instruments. Assuming fixed-rate readers and writers (as in this example), the total delay time possible is the length of the line in frames divided by the sample rate.

The delay time is set by the distance between the attachment points of the writer and reader, specified with the `-startat` option for the *Attach* command. The attachment offset is set so that the player attachment is just a little ahead of the writer (nearly as far away from the writer as possible). This yields the maximum delay time for a given delay line. Because the writer and reader(s) are asynchronous instruments, though, it's generally advisable to leave a buffer zone of around 50 frames between the attachment points for each, so that the writer doesn't advance past the reader and cause a glitch in the output. You should adjust the length of the delay line to compensate.

See “*Examples/Audio/Patches/Reverbs*” for examples of precisely setting delay times and other more sophisticated uses of delays.

flange.mp

Normally, when you attach a sampler instrument to a delay line, you fix the tap point at some offset from the input. However, many useful effects can be achieved through precisely varying the tap point, such as flanging, pitch shifting, and so on. Unfortunately, because the instrument writing into the delay line and the instrument reading from the delay line are independent, it's not possible to do this precisely using the standard sampler instruments.

This patch introduces a new instrument, `sampler_drift_v1.dsp`. This instrument has a port called *Drift*, that has a value of 0 when the sampler is marching through one sample at a time. At other frequencies, when the sampler is interpolating, *Drift* ramps up or down, indicating how many samples away from the *nominal* point we are. By hooking the sampler to a delay, and modifying the sample rate, we are effectively increasing or decreasing the tap reader-writer distance. We use the *Drift* value to precisely control this change; we set an *offset* (the distance at which we want the tap to be set), and then vary the sample rate continuously until (*drift*-*offset*) equals 0. By driving the offset with a continuously varying source (in this case, a triangle lfo), we end up continuously modulating the tap distance, and the result is some degree of pitch-shifting up and down at the output of the sampler. In small degrees, when mixed with the original signal, the result is the comb-filtered *flanging* effect we all know and love. With larger modulation amplitudes, we get serious pitch chorusing, and with faster modulation frequencies we get something akin to FM-y delay.

ramp_frequency.mp

With `envelope.dsp`, you can generate control ramps on the fly, without an attached envelope, by dynamically setting the target toward which to ramp (in the knob `Env.Request`) and the step size to move with (in the knob `Env.Incr`). `Env.Incr` effectively controls the ramp rate—large values correspond to short times. Values at 50 or below are good for smoothing out abrupt volume changes.

In this example, the ramp target is set with a `randomhold.dsp` instrument, so the output of the ramp is moving toward a randomly selected value. The combination of envelope and randomhold could be replaced with a `rednoise_lfo.dsp` instrument, which is a more efficient way of achieving the same end.

ufo.mp

This sound effect patch demonstrates the use of musical instrument samples in a completely non-musical application. This technique yields somewhat more organic and cleaner sounds than you would get using simple synthetic oscillators.

The patch consists of two components: The low pulsating drone is an oboe sample slowed down to about 1/10th of the original sample rate with very slow amplitude modulation. The higher frequency sound is a clarinet sample with a large amount of vibrato, which is run through a high resonance, band-pass filter (`svfilter.dsp` using the `BandPass` output) with a randomly generated cutoff frequency.

The random cutoff frequency is controlled by a combination of `randomhold.dsp` and `envelope.dsp`. This gives a “ramp-hold-ramp-hold” sort of effect. Using `rednoise_lfo.dsp` would result in a constantly varying, “ramp-ramp-ramp-ramp” control signal.

There are a few knobs to control the patch dynamically. There are a pair of rate knobs to control the vibrato rate of the clarinet sample (`Vibrato-Rate`) and the pulsation rate of the low drone (`Drone-ModRate`). The other knobs control the output mix of each of the components.

Reverberation Examples

The directory *Examples/Audio/Patches/Reverbs* contains a selection of reverberator patches, running the gamut from low-overhead/cheesy to high-overhead/medium-quality. To use these patches:

- ◆ Run the script `makepatch.script`, which uses the shell program `makepatch` to generate binary files for each of the examples.

You can test an effects patch by chaining together patches in `dspfaders`. For example:

```
dspfaders impulse.dsp multitap.patch
```

You can also use the `dspfaders -LineIn` switch to send the 3DO audio line input through an effects patch. For example:

```
dspfaders -LineIn multitap.patch
```

The reverbs are described as follows.

1tap.mp: Single tap reverberator

The simplest reverb is a delay line with a single, fixed tap, recirculating some of the output back into the delay. The *dry* and delayed signals can be mixed into the output. See Figure 7-1.

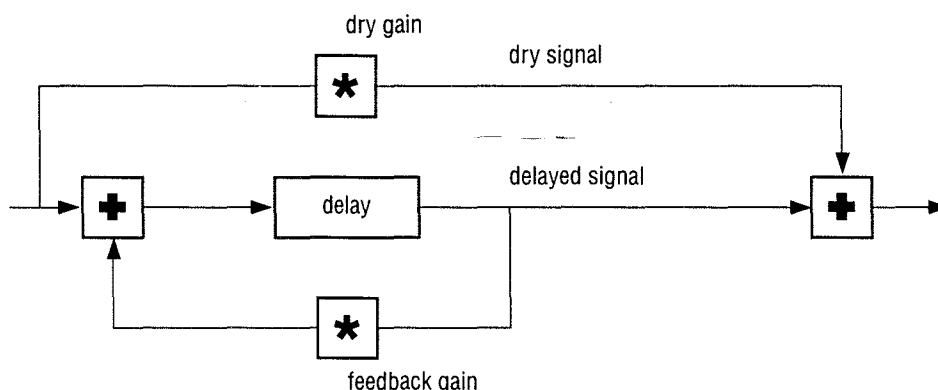


Figure 7-1 1tap.mp

This is so basic that it's almost not *reverb*, but it's easy. It's also highly frequency dependent; the delayed signal forms a comb filter (the basis of a flanging effect), when the delay is constantly changing.

To calculate the length of the delay line required (assuming a feedback gain of about 0.7):

$$\text{totaltime} = 20 * \text{loop delay}$$

Where loop delay is the length of time it takes to get through the delay. At a sample rate of 44100, it takes a delay line of $44100 * 2$ (bytes) for a one-second loop delay. By positioning the output tap a little *ahead* of the input, the signal has to go all the way around the circular delay before being read out. That means setting `-StartAt` to 16 implies a maximum-length delay. Because the input and output aren't necessarily synched, though (due to the nature of the sampler instrument), it's good to separate the input and output pointers in the delay line so they don't cross one another and glitch.

multitap.mp: Multiple tap reverberator

To get denser reverberation, we can use more than one tap off of the delayline, mixed back into the *wet* signal. Ideally, the taps should be spaced so as not to reinforce or cancel each other. In this example, they're spaced at 50, 56, 61, 68, 72 and 78 ms. Figure 7-2

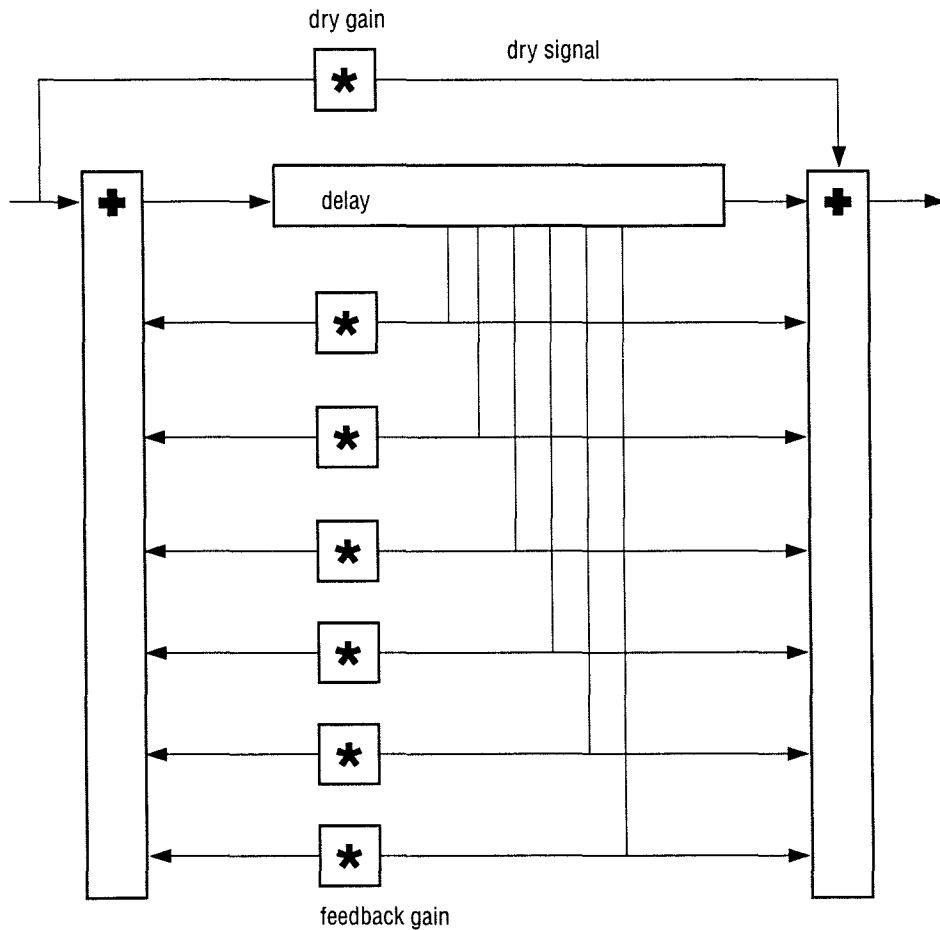


Figure 7-2 *multitap.mp*

Different delay lengths create different comb filter responses, ameliorating the harmonic effect of the single tap somewhat.

lowpass_comb.mp: Three-tap delay line, each with a low-pass filter

Filtering the basic multi-tap reverberator provides a way of modifying the basic comb response of the tap. For simplicity of control, we use an *svfilter* as the filter element, which is somewhat expensive and not too stable (try cranking the feedback gain, Q and sweeping the filter frequency around for pretty nice feedback delay effects). *Nicer* reverberators (in the files suffixed "1o1z") use a simple averaging low-pass filter instead. See Figure 7-3

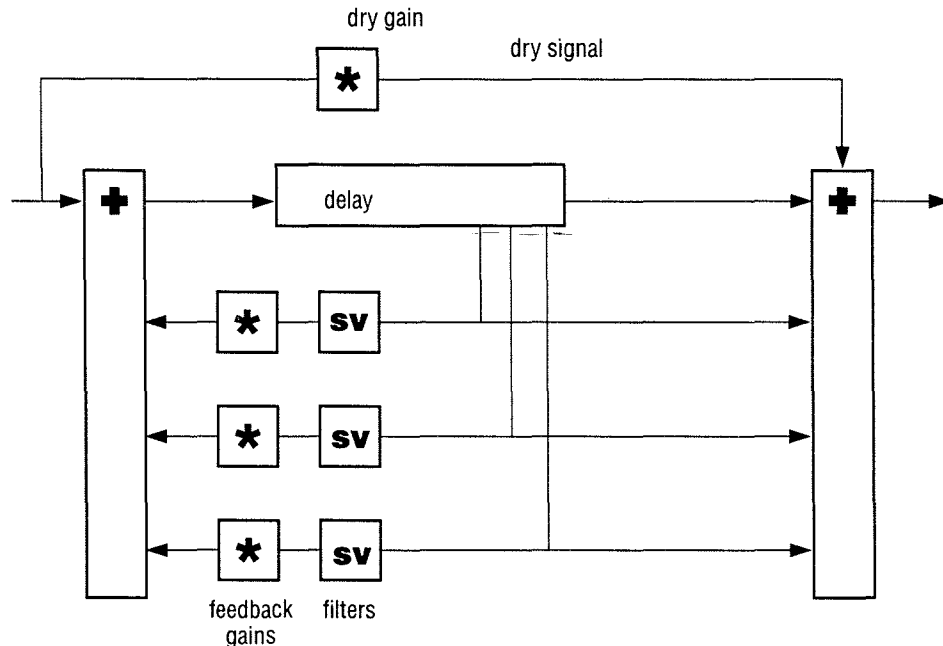


Figure 7-3 *lowpass_comb.mp*

multi_lowpass.mp: Multiple low-pass comb filters, each with their own delay line

Tapping all of the signals off a single delay line just doesn't sound as good as using separate delays for each filter, because of the interaction of the signals. This version uses a small, separate delay for each filter element, forming a classic comb filter. We keep the svfilters in the feedback loops. The delays are sized slightly differently to give the varying delay times, rather than positioning taps at different points.

According to the references, less than six elements makes an audible difference in sound quality. See Figure 7-4.

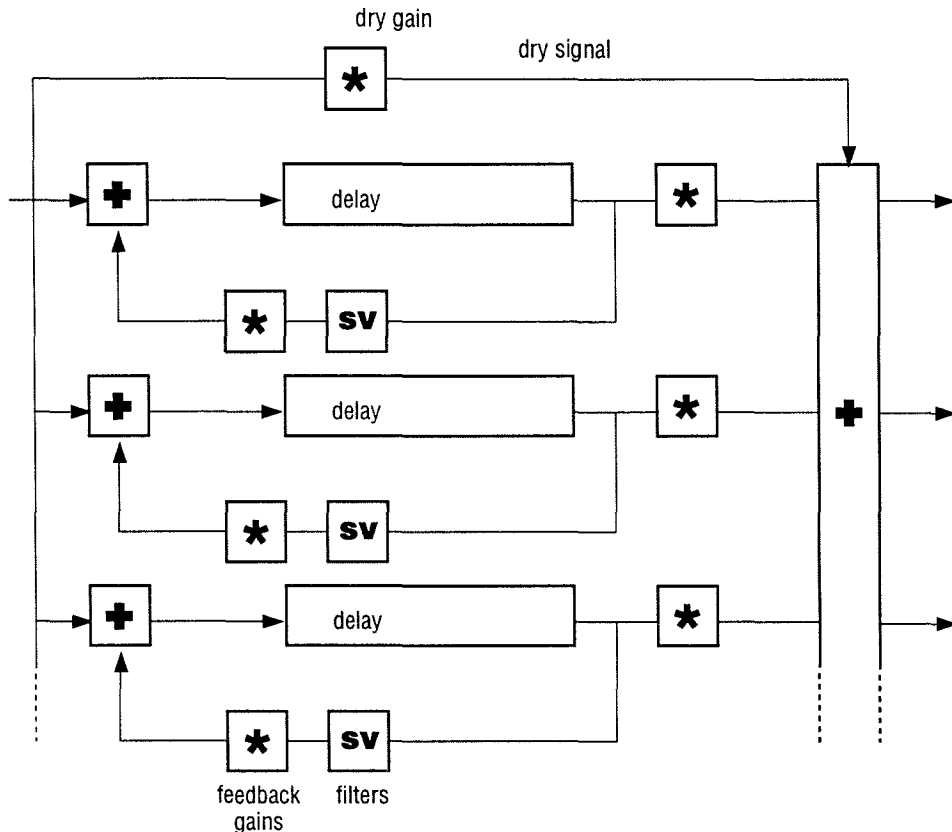


Figure 7-4 `multi_lowpass.mp`

multi_lowpass_101Z.mp: Multiple first-order one-zero low-pass filters, each with their own delay line

This is `multi_lowpass`, with the first-order, 1-zero (1O1Z) filter substituting for the `svfilter` element (refer to Figure 6-4). Not only is it cheaper, but in this configuration it gives a nice, stable, low-resonance behavior that degrades much more gracefully than the `svfilter` (if you prefer grace to raw beauty). Instead of *Frequency* and *Q* controls, we have the filter coefficients *A0* and *A1*, which substitute into the filter equation as follows:

$$y(n) = A0.x(n) + A1.x(n-1)$$

$$x(n) = \text{input at } n, x(n-1) = \text{input at } n-1$$

Setting *A1* to a positive number gives a low-pass characteristic. We set the default values to *A0*=0.5 and *A1*=0.4.

Changing the A0/A1 knobs gives *warmer* or *brighter* reverb character; note that there's interaction with the loop gain g in controlling overall reverb decay time.

allpass_filter.mp: All-pass filter

Unlike the comb and lpcomb filter examples, the all-pass filter has a flat frequency response. It does, however, have a frequency-dependent phase response, so an input signal gets smeared in time. This functions nicely as the final stage of a complex reverb to smooth over artifacts. In addition, by using one all-pass filter on each output channel with slightly different settings, the outputs for each channel are smeared differently. See Figure 7-5

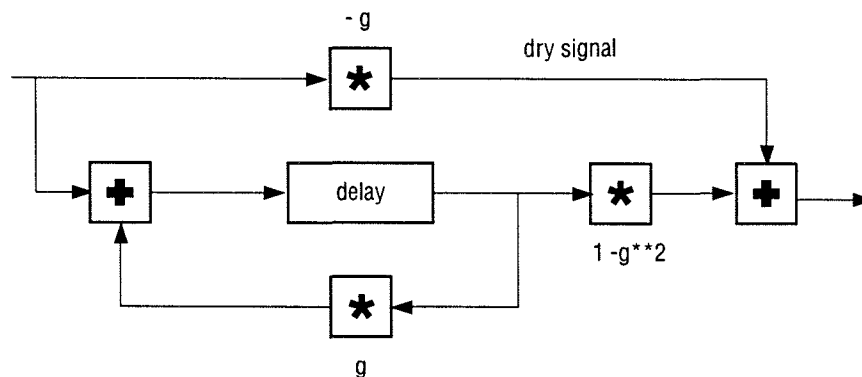


Figure 7-5 *allpass_filter.mp*

As you can see, the gains are all correlated. This patch includes a control path subpatch that, given the value g , outputs g , $-g$, and $1-g^{**2}$ into the appropriate mixer channels (see Figure 7-6).

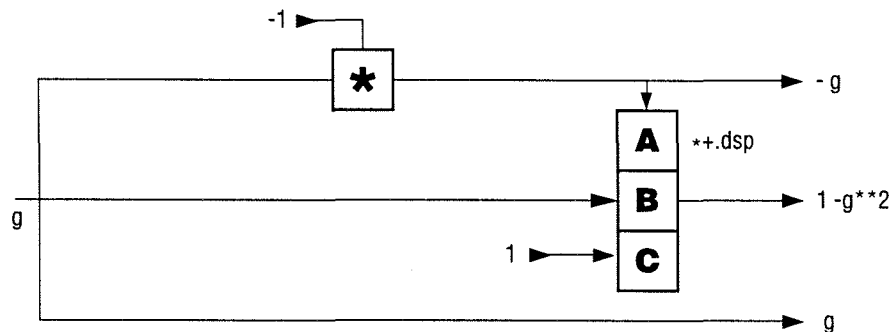


Figure 7-6 *allpass_filter.mp Control Path Subpatch*

multi_lowpass_allpass.mp: Multiple low-pass comb, each with their own delay line, with an all-pass on the mixed output.

Here we've just added the all-pass filter (in `allpass.mp`) to the comb bank in `multi_lowpass.mp`.

multi_lp_allp_101Z.mp: Multiple 101Z, each with their own delay line, with an all-pass on the mixed output.

In this, the *final* reverb, we replace the `svfilter` in `multi_lowpass_allpass` with the first-order 1-zero filter.

3D Sound Spatialization

Introduction

This chapter describes the new 3D Sound which is implemented as part of the 3DO M2 music library. 3D Sound allows you to take an arbitrary mono sound source (a sample, sound file, synthetic sound, or patch) and position it in three dimensional space and then move it around in real time. You use library calls to locate the sound as though it is coming from the point you specify. You can select how much processing to do (and consequently how much of the DSP resources to use) on a sound-by-sound basis. You can “animate” as many sounds in this way as your resources will allow. This generally means 5 or 6 sounds and a sound track. The sound effect is best heard using head phones.

Chapter Overview

This chapter discusses the following topics:.

Topic	Page Number
Introduction	117
Sound Cues	117
Using 3D Sound	120

Sound Cues

3D Sound provides seven cues that can be divided into three groups: positional (left/right, up/down), distance (how far or near a sound seems), and environmental (whether or not the sound is heard inside of a closed space). Each cue, alone or in combination, requires differing amounts of DSP resources and has greater or lesser impact on the illusion of a three dimensional sound. 3D Sound

allows you to determine which combination of cues to use for any given sound so that you can optimize your resources by specifying a set of flags when you create the sound.

Amplitude Panning

Panning is like the balance knob on your home stereo. Panning changes the left-to-right direction that the sound appears to be coming from by altering the amplitude of the sound sent to each ear. Making changes to a sound with amplitude panning has very low DSP overhead.

Delay Panning

There will be a difference in the amount of time a sound takes to reach your left ear and the time it takes to reach your right ear, depending upon where a sound comes from. By artificially recreating that delay, you can change the apparent direction of the sound. Using delay panning rather than amplitude panning sounds more natural, but uses more DSP resources. You can use both cues together to increase the effect to the point that the sound becomes artificial.

Filtering

Filtering cuts out some frequencies and changes the timbre of a sound. An example in speech is the difference between:

ahh

and

ehh

Treble and bass controls on a stereo are simple filters.

3DSound uses a filter to approximate the effects of your ear shape, your head and upper body on a sound. We also account for the loss of high frequencies with distance. This increases the realism of either amplitude panning or delay panning; makes a sound appear in front of you, behind you, above you or below you, and contributes to the perception of distance. The 3D Sound Filter works like the Binaural Filter of SoundHack described in the *3DO M2 Tools for Sound Design* manual.

Doppler

The doppler cue tells you how much to alter your source sound's parameters (usually frequency, sometimes others) to make it seem as if the sound is coming toward you or receding from you. You only use this cue if the sound is moving relative to your listener. Dopplering takes no DSP resources, but increases the overhead for 3DSound movement function calls.

Distance Factor

In real life, sounds get quieter as they move farther away from you. You can simulate this gradual quieting by scaling the gain of the sound before it's sent to the output. The distance factor cue tells you how much to scale the gain.

There are two distance factor calculations available: `distance_square`, and `sone`. The first calculation works best for most sounds, especially when combined with the filter cue. Use the `sone` calculation for a more pronounced effect, which seems most natural with familiar sounds. — —

You can also use the distance factor to mix reverberation and echo in with the 3D sound. When you hear a sound, first it arrives directly from the source. Then, you hear a few distinct echoes from reflections off walls or other objects in the space. Finally, you hear an amalgam of diffuse reflections called a reverberation. While the 3DSound library won't implement these effects for you, you can use the distance factor cue and one of the Reverb patches in the directory `Examples/Audio/Patches/Reverbs` to approximate the effects for yourself.

Directionality

Some objects emit sound equally in all directions, for example, a tree rustling in the wind. Some objects emit sound more strongly in one direction than in another direction like a trumpet or a loudspeaker. This directionality of a sound can be represented by a number between 0.0 and 1.0, where 0 says that no sound is emitted from the back of the object. If the object points away from you, then you cannot hear it. When directionality is set to 1.0, the same amount of sound issues from the front and back of the object.

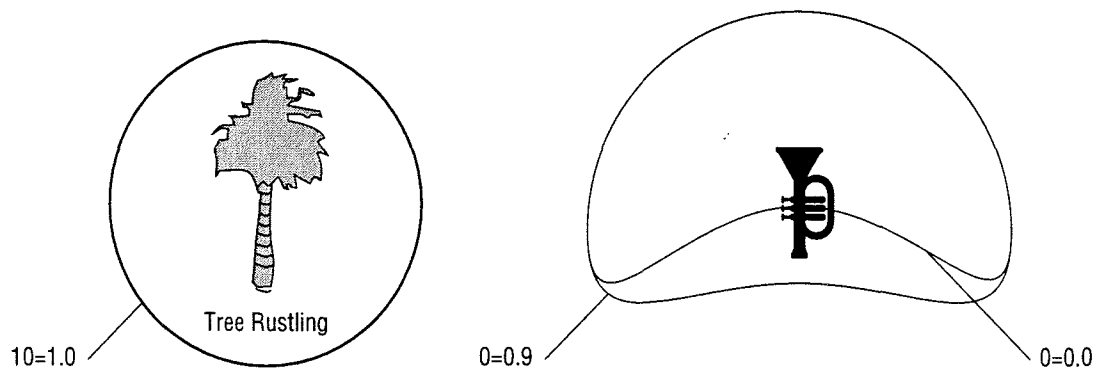


Figure 8-1 Rustling Tree and Trumpet for 3D Directionality Examples

Figure 8-1 shows a tree example, where the direction of sound appears to be *everywhere*. The trumpet example shows what the sound looks like when its

directionality is equal to 0.0 and there is no sound emitted from behind the trumpet. When Directionality is set to 0.9, the loudness of the sound emitted behind the trumpet is a little less than the loudness of the sound in front.

Locating 3D Sound Examples

There are three sound examples included in the Examples/Audio/Sound3D directory: ta_steer3d, ta_bee3d, and ta_leslie. There is an additional graphic-driven version of ta_bee3d in the subdirectory SeeSound that shows three objects in space, lets you hear them and move your own location relative to them. Most of these examples include reverberation as described here.

Using 3D Sound

A sound in 3D space has three components: a source instrument, a 3DSound and an environment. The source instrument is the monophonic, unspatialized sound. It should be as "dry" as possible (that is, no reverberation already added). For each source, there's a corresponding 3DSound. Each 3DSound's stereo output will be connected to the environment. The sound environment may be as simple as a stereo mixer, or it may include additional effects like reverberation.

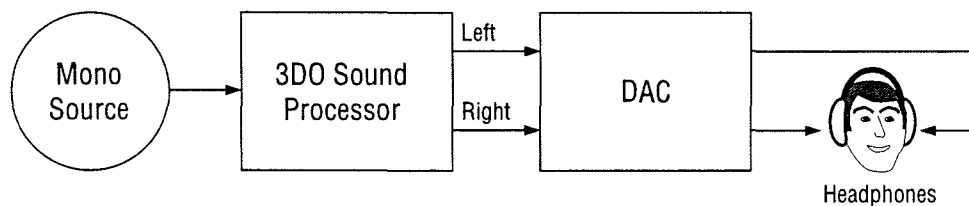


Figure 8-2 *Simplest case of a stereo mixer for 3D sound*

Other sounds, like a sounders or non-spatialized sound effects might also be mixed into the environment.

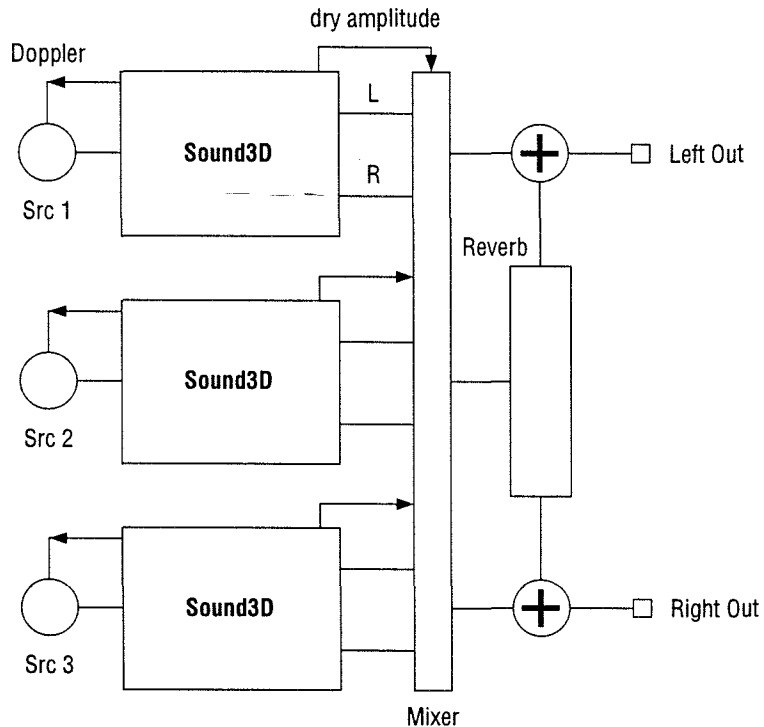


Figure 8-3 3D Sound Space

Setting up 3D Sounds

After creating the source sound and environment instruments, the application creates a 3DSound object by calling `Create3DSound`. The application passes a pointer to a variable that is set by `Create3DSound` to point to the 3DSound structure created, and an array of tag arguments. The tags specify which cues to use for the sound, and (for directional sounds) the back amplitude.

Next, the application needs to interconnect the sonic components. A 3DSound object contains a custom instrument to implement the spatialization, to which the source sound needs to be connected. To get the item number of the 3DSound instrument, the application uses the function `Get3DSoundInstrument`. The instrument has one input port, "Input", and a 2-part stereo output port, "Output". The application connects the output of the source instrument to the "Input" port of the 3DSound instrument and the "Output" port of the 3DSound to the environment.

Pseudo Code example of 3D Sound

A typical application might use 3DSound something like the pseudo-code description that follows.

Example 8-1

```
set up source instrument, environment
Create3DSound()
3Dinst = Get3DSoundInstrument()
connect source instrument, 3Dinst and environment instrument
start environment
Start3DSound()
start source instrument
loop
    calculate new positions
    Move3DSound()
    Get3DSoundParms
    modify host-level parameters accordingly
stop source instrument
Stop3DSound()
stop environment
Delete3DSound
kill off source instrument, environment
```

Starting 3D Sounds

Next, the components are started. As usual, instruments further down the chain (closer to the final output) are started before ones further up to avoid missing the attack transients of the sound. First the environment should be started, followed by a call to `Start3DSound`. Finally, the source instrument is started. `Start3DSound` takes as arguments the pointer to the 3DSound structure set by `Create3DSound`, and a starting position.

Position in the 3DSound library is specified using a structure type called a `PolarPosition4D`, defined in the header file `sound3d.h`. The structure has four components: a radius, azimuth angle (theta) and elevation angle (phi) that specify the position in polar coordinates and a time at which the position applies. The radius is specified in frame units, which are defined as the distance that sound travels in one audio frame tick (1/44100 seconds). Under normal circumstances, this is approximately 340meters/second, the speed of sound in air at sea level. The azimuth is defined in radians, relative to an axis straight ahead of the listener. The angle is positive to the right, and negative to the left. Elevation is

also in radians, relative to the horizontal plane. Positive is up, negative is down. Time is in audio frame ticks.

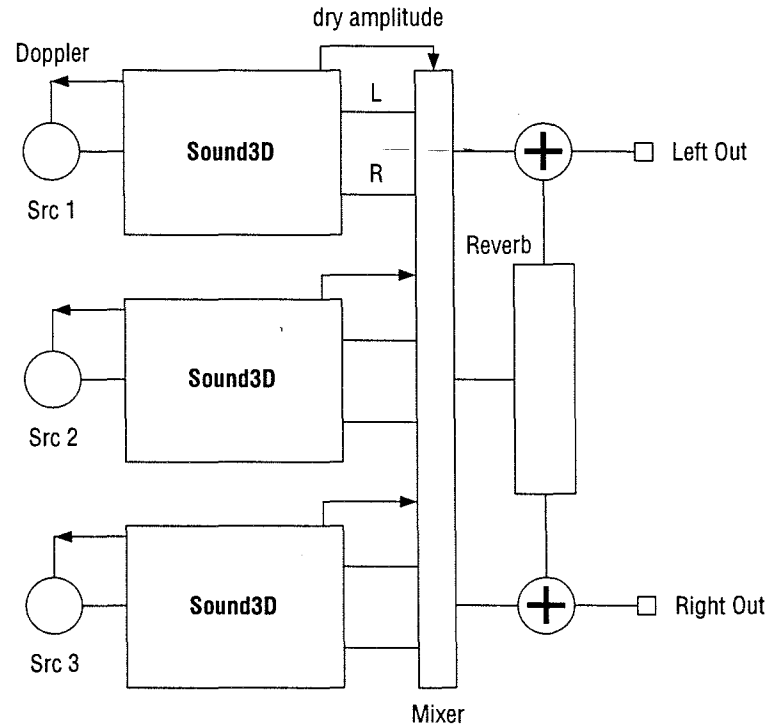


Figure 8-4 The polar axis for *PolarPosition4D*

After a call to `Start3DSound()` or `Move3DSound()`, an application may need to update parameters of the source instrument or the environment based on position information. To do this, the application calls `Get3DSoundParms()`, which copies the distance factor and the doppler information into an application-allocated `3DSoundParms` structure. The application then uses this information to, say, change the source sound's frequency to approximate doppler shift and change the gain and wet/dry reverberation mix in the environment.

At this point, the application is producing an audible sound that is positioned in three dimensions.

Moving 3D Sounds

A 3D Sound can move relative to the listener in a straight line over a short period. To make a sound move in something other than a straight line, or move over a longer period, the trajectory is decomposed into a sequence of short line

segments, and the movement is done in a loop, much like graphics animation. In fact, the 3DSound movement calls are intended to be embedded inside the main loop of such a program in the same way that the graphics rendering calls are.

For each frame:

- ◆ calculate new positions
- ◆ move the sound(s)
- ◆ render the graphics

The main function used to move a sound is `Move3DSound`. It accepts a pointer to the 3DSound, and two `PolarPosition4D` pointers: a starting point, and an ending point. The sound moves from the starting point to the ending point at a rate determined by the difference between ending and starting times.

Typically, an application moves a sound from wherever it is at the time the call is made to a specified position. To make this easier, the application can use the function `Move3DSoundTo`. This function uses the current position and time of the sound, as returned by the function `Get3DSoundPos`, as the starting point.

Note: *the doppler cue is particularly dependent on accurate time values, as it is proportional to a sound's velocity. If you find that your doppler values are wiggling around too much when you're using `Move3DSoundTo` because your loop timing isn't stable, try using `Move3DSound` with constant time increments.*

After each call to `Move3DSound` or `Move3DSoundTo` you should update host-level parameters. Refer to the section on Starting 3D Sounds for more information on this.

Deleting 3D Sounds

An application stops a 3D Sound with a call to `Stop3DSound`. The application might reuse the 3D Sound with another source instrument, or get rid of it. This latter action is performed using a call to `Delete3DSound`, which just takes as a parameter the pointer to the 3DSound structure set with `Create3DSound`. Deleting a 3D Sound stops it first, and breaks all the connections to the source instrument and environment.

Sound Player

This chapter describes the Sound Player, the Music Library component for spooling AIFF sound files. This chapter contains the following topics:

Topic	Page Number
An Overview of the Sound Player	125
Players, Sounds, and Markers	128
Decision Functions	130
Caveats	134
For More Information	134

The previous chapters describe low-level calls that allow you to load a sampled sound file into memory as a sample item. You can then play back the sample with a sampled sound instrument. This process limits the sampled sound playback to whatever you have loaded into memory. If you have a very large sampled sound file and limited memory, then you can only play back one part of the sampled sound before you must stop, load a new section of sampled sound from the file, and then play the newly loaded section. To solve the problem of segmented sound file playback, the Music library offers the Sound Player.

An Overview of the Sound Player

The idea behind the Sound Player is similar to the idea behind double buffered animation: provide separate buffers for sampled sound playback, then play back the sound buffers in continuous sequence as shown in Figure 9-1. While one buffer plays, the Sound Player can safely write more of the sound file into buffers

not being played. Because the spooler need only write to buffers at intervals (and not continuously), glitches caused by interrupted or slow disc access are unlikely. Playback continues smoothly from buffer to buffer without pause.

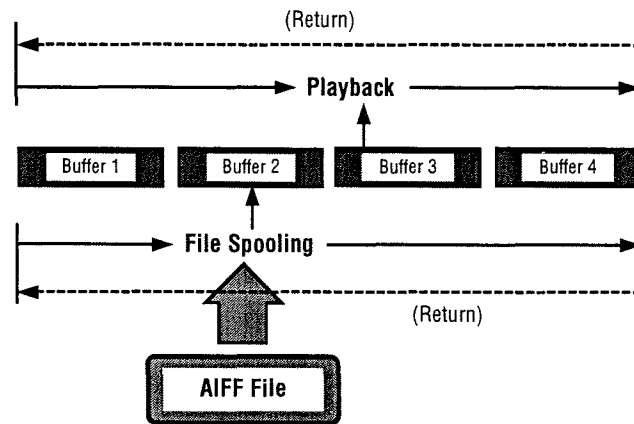


Figure 9-1 *Playing sounds in a continuous sequence*

The Sound Player is build on top of a general Sound Spooler. See Chapter 10, "Using the Sound Spooler" for a description.

Note: *The Sound Player works better spooling a sound file from CD than it does spooling a sound file from the Macintosh in a development system because the file system on the CD uses DMA.*

In addition to simply spooling an AIFF sound file from disc, the Sound Player also does the following:

- ◆ Permits sound files to be looped seamlessly (no audible pops or silent gaps).
- ◆ Allows multiple sound files to be linked together seamlessly.
- ◆ Permits branching between markers in sound files.
- ◆ Allows playback of in-memory samples as if they were sound files.

Simple Sound Player Example

This annotated Sound Player example demonstrates how to play an AIFF sound file through once. Error checking is omitted for brevity. More complicated examples are built from this one later in this chapter.

1. Preparation. Note that this step makes a number of assumptions about the kinds of sounds that are going to be played and how to direct the output signal to the 3DO audio output. In this case, we are expecting to play a monophonic 16-bit 44KHz sample and are connecting it directly to the audio

output. In practice, you may need to select a different sample player instrument or an alternate method of directing the output.

```
samplerIns = LoadInstrument ("sampler_16_f1.dsp", 0, 100);
outputIns = LoadInstrument ("line_out.dsp", 0, 100);
ConnectInstrumentParts (samplerIns, "Output", 0,
                        outputIns, "Input", AF_PART_LEFT);
ConnectInstrumentParts (samplerIns, "Output", 0,
                        outputIns, "Input", AF_PART_RIGHT);
StartInstrument (outputIns, NULL);
```

2. Create a player.

```
spCreatePlayer (&player, samplerIns, NUMBUFFS, BufSize, NULL);
```

3. Specify an AIFF sound file to play.

```
spAddSoundFile (&sound, player, filename);
```

4. Set up branches. In this example, there are no branches to set up, so this step is simply a placeholder. "Examples of Looping and Branching" on page 129 shows how to modify this example to do more interesting kinds of playback.

5. Start reading sound. This step prefills the spooler with sound data so that the next step can begin producing sound immediately.

```
spStartReading (sound1, SP_MARKER_NAME_BEGIN);
```

6. Start playing sound.

```
spStartPlayingVA (player,
                  AF_TAG_AMPLITUDE_FP, ConvertFP_TagData(1.0),
                  TAG_END);
```

7. Service the player by receiving its signals and calling `spService()`. Stay in this loop until sound playback is complete.

```
const int32 playersigs = spGetPlayerSignalMask (player);

while (spGetPlayerStatus(player) & SP_STATUS_F_BUFFER_ACTIVE)
{
    const int32 sigs = WaitSignal (playersigs);
    spService (player, sigs);
}
```

8. Clean up. Note that `spDeletePlayer()` takes care of cleaning up anything that was given to it (e.g. the sound file added in step 3). The cleanup process

is more completely described in the *3DO Music and Audio Programmer's Reference*.

```
spStop (player);  
spDeletePlayer (player);  
UnloadInstrument (outputIns);  
UnloadInstrument (samplerIns);
```

Players, Sounds, and Markers

The following material describes sound player objects and how to do simple looping and branching, together with examples.

Players

The top-level data structure used by the Sound Player is called an `SPPlayer`. Each `SPPlayer` contains a sound spooler, a sample player instrument, a set of buffers, and control information necessary to manage playback. A task can have multiple `SPPlayers`, as long as they do not compete for the same media (e.g., two or more `SPPlayers` trying to play sound from a CD at the same time is not likely to work).

Sounds

`SPPlayers` also contain a set of `SPSounds` they can play. An `SPSound` is a handle to sound data source. Currently two kinds of sound sources are supported: AIFF sound files and sample items. The difference is that AIFF sound files are spooled directly from disc and sample items are played entirely from memory. Since `SPSounds` built around sample items do not require access to the CD during playback, they can be used to keep playing sound while accessing the CD for something else. Other than that, different kinds of `SPSounds` are treated identically by the Sound Player.

The set of `SPsounds` belonging to an `SPPlayer` can be managed at any time, even during playback. All of the `SPSounds` belonging to an `SPPlayer` must have the same sample format: number of channels, sample width, compression type and ratio.

Markers

Each `SPSound` can have a set of markers at key sample positions within the sound data. These `SPMarkers` can be defined in the following ways:

- ◆ Markers and loop points from an AIFF file.
- ◆ Loop points from a sample item.
- ◆ Begin and end markers (intrinsic `SPMarkers` created by the Sound Player for all `SPSounds`).

- ◆ SPMarkers created by the client.

Any of the SPMarkers can be set to cause playback to branch seamlessly (without any audible pops or clicks) to another SPMarker within the same SPSSound or any of the other SPSSounds that have been added to the SPPlayer.

By default, SPMarkers do nothing during playback. Also, by default, when the end of the sound data for the currently playing SPSSound is reached, playback automatically stops. This can be changed by the client by correct placement of branches either before or during playback. The following examples illustrate a few things that can be done with markers.

Examples of Looping and Branching

Here are some simple modifications to the previous annotated example to do some simple branching.

Play Several Sound Files Sequentially

Substitute these steps in the annotated example to link several sounds together for seamless playback.

3. Specify which sound files you want to play.

```
spAddSoundFile (&first_sound, player, filename);
spAddSoundFile (&second_sound, player, filename);
spAddSoundFile (&third_sound, player, filename);
.
.
.
spAddSoundFile (&penultimate_sound, player, filename);
spAddSoundFile (&final_sound, player, filename);
```

4. Set a branch at the end of each sound to the beginning of the next one. Each sound has intrinsic markers for its beginning and ending.

```
spBranchAtMarker (first_sound, SP_MARKER_NAME_END,
                  second_sound, SP_MARKER_NAME_BEGIN);
spBranchAtMarker (second_sound, SP_MARKER_NAME_END,
                  third_sound, SP_MARKER_NAME_BEGIN);
.
.
.
spBranchAtMarker (penultimate_sound, SP_MARKER_NAME_END,
                  final_sound, SP_MARKER_NAME_BEGIN);
```

Because there is no branch at the end of the final sound, playback stops when the end of the final sound is reached.

There is also a convenience macro for linking multiple sounds together that does the above `spBranchAtMarker()` calls:

```
spLinkSounds (first_sound, second_sound);
spLinkSounds (second_sound, third_sound);
.
.
.
and so on.
```

Play a Sound File Continuously in a Loop

Substitute this step in the annotated example to create a seamless loop for the sound file and play that loop continuously.

4. Set a branch at the end of the sound back to the beginning of the sound.

```
spBranchAtMarker (sound, SP_MARKER_NAME_END,
                  sound, SP_MARKER_NAME_BEGIN);
```

There is also a convenience macro for this:

```
spLoopSound (sound);
```

Note that this example does not stop automatically because the branch at the end of the sound always goes back to the beginning of the same sound. The sound file continues to loop until you instruct it to stop. One way to do that is to call `spStop()` to stop the playback immediately, but this will more than likely cause an audible pop. To avoid this while still stopping almost immediately, you could use an envelope to ramp down the amplitude of the sample player instrument prior to calling `spStop()`. You can also cause playback to stop the next time the end of the sound is reached by clearing the branch at the end of the sound:

```
spStopAtMarker (sound, SP_MARKER_NAME_END);
```

You can do this at any time while playing the sound.

Decision Functions

So far, this chapter has described how to perform primarily static actions. These actions include the ability to instruct playback to always branch, continue, or stop at any given `SPMarker`. Any `SPMarker` can have precisely one static action at any time. This can be changed dynamically during playback, but in order to do this intelligently, it is important to know precisely *when*.

The Sound Player provides a dynamic action facility known as decision functions. A decision function is a callback function that the Sound Player calls during playback in order to determine what action to take at selected `SPMarkers`.

Before going into more detail, it is important to understand when decisions about sound playback need to be made. Sound data is read from whatever sound source it comes from (file or memory) and then enqueued to a sound spooler, which is in turn played by the Audio folio. Because of the spooling, there is an certain amount of delay between the time that sound is read and when it is actually played. All decisions about where to branch, or when to stop, take place while *reading* the sound. Branches do not cause the sound data in the spooler to be flushed, they merely cause the reader to read from a different location in the sound data. Therefore, all decisions have a certain amount of latency before the results are actually heard.

Note: *Since the amount of latency is directly related to the size of the sound buffers you supply to the Sound Player, you have some important design decisions to make when writing a program that uses the Sound Player. See the Caveats at the end of this chapter.*

Decision functions are called when selected SPMarkers are encountered while reading sound data. There are two kinds of decision functions:

- ◆ Marker decision functions
- ◆ Default decision functions

Each SPMarker can have its own marker decision function installed, in addition to whatever static action it may have. An SPMarker's decision function is called when that SPMarker is encountered while reading. Each SPPlayer can have its own default decision function which is called for every SPMarker encountered while reading.

Just like static actions set in an SPMarker, decision functions can return an action to perform: branch to another SPMarker or stop reading. Also, decision functions have the ability to return a *don't care* result, which causes the Sound Player to act as if the decision function had never been installed.

The following pseudocode shows the steps that are used in determining what action the Sound Player takes at an SPMarker:

```
if SPMarker has a decision function and that decision function specifies an action
    take action specified by decision function
else if SPPlayer has a default decision function and that decision function specifies an action
    take action specified by decision function
else if SPMarker has a static action (branch or stop)
    take action specified in the SPMarker
else if playback has reached the end of the current sound data
    stop
```

else

continue playback after the current SPMarker

Decision Functions and Actions

Decision functions are functions whose type can be cast to `SPDecisionFunction`. This is defined in `<:audio:soundplayer.h>` as follows:

```
typedef Err (SPDecisionFunction) (SPAction *resultAction,
                                  void *decisionData,
                                  SPSSound *sound,
                                  const char *markerName)
```

Decision functions are called with a client supplied data pointer, information about the SPMarker at which the decision is to be made, and an SPAction to fill out.

SPAction is a black box data type that is used to contain the action that the decision function wishes the Sound Player to take. The decision function can call either `spSetBranchAction()` or `spSetStopAction()` to set the SPAction to instruct the Sound Player to branch to another SPMarker or stop reading. Alternatively, the decision function can simply not set the SPAction at all. In this case, the Sound Player continues as if the decision function had not been installed. This can be used to permit whatever static action is set up in the SPMarker to normally take place, but to override this occasionally with some other action from the decision function.

Example Decision Function

Here are some simple modifications to the previous annotated example to do some simple branching.

Loop a Sound File a Fixed Number of Times

Substitute this step in the annotated example to play a sound file four times in a loop. This is excerpted from `tsp_spoolsoundfile.c`.

4. Set up sound to loop by default. Install a Decision Function at the end SPMarker to be called each time playback reaches the end of the sound.

```
spLoopSound (sound);
RemainingCount = 4;    /* play 4 times */
spSetMarkerDecisionFunction (
    sound, SP_MARKER_NAME_END,
    (SPDecisionFunction)LoopDecisionFunction,
    &RemainingCount);
```

The decision function, `LoopDecisionFunction()`, looks like this:

```
/*
   This decision function decrements the RemainingCount variable.
   Normally it does nothing. When the count is exhausted, it
   instructs the Sound Player to stop reading.
*/
Err LoopDecisionFunction (
    SPAction *resultAction, int32 *remainingCountP,
    SPSound *sound, const_char *markerName)
{
    /* decrement remaining repeat count */
    (*remainingCountP)--;

    /* loop back to beginning (default action) or stop if
       no more repeats */
    if (*remainingCountP <= 0)
        return spSetStopAction (resultAction);
    else
        return 0;
}
```

Rules for Decision Functions

Besides setting an `SPAction`, a decision function can do almost anything to the `SPPlayer` that owns this marker including:

- ◆ adding new `SPSounds` or `SPMarkers`
- ◆ changing the static action for this or any other `SPMarker`
- ◆ changing the default or marker decision functions for this, or any other, `SPMarker`
- ◆ deleting this, or any other, `SPMarker` or `SPSound` (with the caveats listed below in mind).

A decision function *must not* do any of the following:

- ◆ Call any function that changes the player state (e.g. `spDeletePlayer()`, `spStop()`, `spStartReading()`, `spStartPlaying()`, `spService()`, etc.) for the current `SPMarker`'s `SPPlayer`.
- ◆ Delete the current `SPMarker`'s `SPSound` since this has the side effect of calling `spStop()`.
- ◆ Delete the current `SPMarker` without setting up `resultAction` to stop or branch to another `SPMarker`.
- ◆ Take a long time to execute.

Also note that because decision functions are called on the reader's schedule, not the player's schedule, they cannot be used for notification that a particular part of the sound has actually been played.

See the description of the `SPDecisionFunction` in the *3DO Music and Audio Programmer's Reference* for more complete information.

Caveats

The application programmer controls how well the Sound Player functions. You should keep the following points in mind when writing programs that use the Sound Player:

- ◆ Branches must make sense in terms of the sound data being played back. The Sound Player simply switches from one sound source to another. It does not crossfade.
- ◆ The service function, `spService()`, must be called frequently enough to avoid spooler starvation. Spooler starvation causes silent gaps in the sound output.
- ◆ The spooler buffers must be sufficiently large to avoid starvation while performing the branches that are called for, yet be small enough to keep branch response time reasonable.
- ◆ Be careful when using default decision functions because they get called at every `SPMarker`. Since this effectively causes the Sound Player to only be able to read and spool data between each `SPMarker` at a time, densely packed `SPMarkers` can have a serious impact on efficient use of disc I/O and the sound spooler.

For More Information

There are complete examples using the Sound Player in the *Examples/Audio/Spooling* folder.

Function Calls

These Music library function calls control the Sound Player. See Chapter 2, "Music Library Calls" in the *3DO Music and Audio Programmer's Reference* for additional information on these calls.

Player Management Function Calls

The following calls are used to manage `SPPlayers`:

<code>spCreatePlayer()</code>	Create an <code>SPPlayer</code> .
-------------------------------	-----------------------------------

<code>spDeletePlayer()</code>	Delete an <code>SPPlayer</code> .
<code>spGetPlayerSignalMask()</code>	Get set of signals player sends to client for an <code>SPPlayer</code> .
<code>spStartReading()</code>	Start <code>SPPlayer</code> reading from an <code>SPSound</code> .
<code>spStartPlaying()</code>	Begin emitting sound for an <code>SPPlayer</code> .
<code>spStop()</code>	Stop an <code>SPPlayer</code> .
<code>spPause()</code>	Pause an <code>SPPlayer</code> .
<code>spResume()</code>	Resume playback of an <code>SPPlayer</code> that has been paused.
<code>spService()</code>	Process signals sent by the <code>SPPlayer</code> , read and spool more sound data, and process markers.
<code>spGetPlayerStatus()</code>	Get playback status of an <code>SPPlayer</code> .

Sound Management Function Calls

The following calls are used to manage `SPSounds`:

<code>spAddSample()</code>	Create an <code>SPSound</code> for a Sample Item.
<code>spAddSoundFile()</code>	Create an <code>SPSound</code> for an AIFF sound file.
<code>spRemoveSound()</code>	Remove an <code>SPSound</code> from an <code>SPPlayer</code> .
<code>spGetPlayerFromSound()</code>	Get <code>SPPlayer</code> that owns the specified <code>SPSound</code> .
<code>spIsSoundInUse()</code>	Determines if <code>SPPlayer</code> is currently reading from a particular <code>SPSound</code> .

Marker Management Function Calls

The following calls are used to add and remove `SPMarkers` or set up a static action at an `SPMarker`:

<code>spAddMarker()</code>	Add a new <code>SPMarker</code> to an <code>SPSound</code> .
<code>spRemoveMarker()</code>	Remove an <code>SPMarker</code> from an <code>SPSound</code> .
<code>spFindMarkerName()</code>	Finds an <code>SPMarker</code> by name belonging to the specified <code>SPSound</code> .

<code>spGetSoundFromMarker()</code>	Returns a pointer to the SPSSound to which the specified SPMarker belongs.
<code>spGetMarkerPosition()</code>	Returns the frame position of an SPMarker.
<code>spGetMarkerName()</code>	Returns the name of an SPMarker.
<code>spContinueAtMarker()</code>	Restores the SPMarker to its default action.
<code>spStopAtMarker()</code>	Stop when playback reaches SPMarker.
<code>spBranchAtMarker()</code>	Set up a static branch at an SPMarker.
<code>spLinkSounds()</code>	Branch at end of one SPSSound to beginning of another.
<code>spLoopSound()</code>	Branch at end of an SPSSound back to its beginning.

Decision Function Calls

The following calls are used to install and remove decision functions and to set SPAction from within decision functions:

<code>spSetDefaultDecisionFunction()</code>	Install a global decision function to be called for every SPMarker.
<code>spClearDefaultDecisionFunction()</code>	Removes a global decision function installed by <code>spSetDefaultDecisionFunction()</code> .
<code>spSetMarkerDecisionFunction()</code>	Install a marker decision function for the specified SPMarker.
<code>spClearMarkerDecisionFunction()</code>	Removes a marker decision function installed by <code>spSetMarkerDecisionFunction()</code> .
<code>spSetBranchAction()</code>	Called from within a decision function to specify that resulting decision is a branch to an SPMarker.
<code>spSetStopAction()</code>	Called from within a decision function to specify that resulting decision is to stop.

Debug Function Calls

The following call is used to display debugging information about an SPPlayer.

<code>spDumpPlayer()</code>	Print debug information for an SPPlayer.
-----------------------------	--

Constants

The following constants are used to refer to intrinsic markers:

<code>SP_MARKER_NAME_BEGIN</code>	Set to the beginning of the sample
<code>SP_MARKER_NAME_END</code>	Set to the end of the sample
<code>SP_MARKER_NAME_SUSTAIN_BEGIN</code>	Set to the beginning of the sustain loop if the sample has a sustain loop.
<code>SP_MARKER_NAME_SUSTAIN_END</code>	Set to the end of the sustain loop if the sample has a sustain loop.
<code>SP_MARKER_NAME_RELEASE_BEGIN</code>	Set to the beginning of the release loop if the sound file has a release loop.
<code>SP_MARKER_NAME_RELEASE_END</code>	Set to the end of the release loop if the sound file has a release loop.

Using the Sound Spooler

This chapter describes the sound spooler function calls and tells you how to incorporate them into your application.

This chapter contains the following topics:

Topic	Page Number
How the Sound Spooler Works	132
How to Use the Sound Spooler	132
An Example	133
Convenience Functions	136
Sound Spooler Function Calls	138

The sound spooler lets you create a custom sound file player or spool sound coming from anywhere other than an AIFF file; for example, a movie file. By including a sound spooler in your application you can spool blocks of sample audio data to the audio digital signal processor (DSP). The sound spooler manages a queue of audio buffers and controls the Audio folio.

Note: *The Sound Spooler is a low-level support system that is used by the Sound Player and Data Streamer. In most cases, you use the existing higher level clients of the Sound Spooler (e.g., Sound Player for AIFF files, or the Data Streamer mixed video and audio files) instead of creating your own. However, if you want to play samples that are too large to fit in memory or play files that must be spooled from disc AND you cannot use existing higher-level systems, you need to create a new player.*

How the Sound Spooler Works

This section provides an overview of the sound spooler and describes how it works. See Chapter 2, "Music Library Calls," in the *3DO Music and Audio Programmer's Reference* for complete descriptions of the calls mentioned.

The sound spooler is part of the Music library that must be linked with your application. It provides function calls that can send blocks of audio data to the Audio folio for asynchronous playback. See "How to Use the Sound Spooler" on page 132 for instructions on how to do this.

To create an instance of the sound spooler call `ssplCreateSoundSpooler()`. You can have multiple sound spoolers running at one time, but if they all get their data from the CD-ROM it could cause disc thrashing, which would affect performance.

The sound spooler has a list of buffer nodes, each created by a call to `ssplCreateSoundBufferNode()`, that manage sampled sound buffers through the spooler. These nodes are created for you automatically when you call `ssplCreateSoundSpooler()`. Each buffer node has a `UserData` field that the application can use to store a pointer to its own per-buffer-node information. The sampled sound buffers themselves are created, filled, reused, and deleted by the application. The application maintains a "free list" and a FIFO "active" playback queue of buffer nodes.

The application calls `ssplSetBufferAddressLength()` and then `ssplSendBuffer()` (or one of the convenience routines) to associate a block of sampled sound data with a buffer node and move it from the free list to the active queue. Buffers in the playback queue are called active.

When an active buffer finishes playing, the Audio folio sends its signal to the application. When the application notices one or more of these signals, it must call `ssplProcessSignals()` to recycle buffer nodes back to the spooler's free list. `ssplProcessSignals()` then calls an optional client call back procedure for each buffer that finished playing.

How to Use the Sound Spooler

In order to use the sound spooler, your application must do the following:

1. Load a sample player instrument and connect it as desired (e.g., connect sample player instrument's output to `line_out.dsp`).

See previous chapters for information on loading a player and instrument. Do not forget to start other instruments.

2. Create a `SoundSpooler` data structure by calling `ssplCreateSoundSpooler()`:

```
SoundSpooler *ssplCreateSoundSpooler (int32 NumBuffers,  
    Item SamplerIns)
```

3. Allocate sampled sound buffers and fill them with the first batch of data from disc or elsewhere. Then spool the buffers into the active queue by calling `ssplSpoolData()`:

```
int32 ssplSpoolData (SoundSpooler *sspl, char *Data,  
    int32 NumBytes, void *UserData)
```

4. Start the sound spooler by calling `ssplStartSpoolerTags()`:

```
Err ssplStartSpoolerTags (SoundSpooler *sspl,  
    const TagArg *startTagList)
```

5. Monitor the signals returned by `ssplSpoolData()`.

If you are using the sound spooler in a loop, you must monitor the signals returned by `ssplSpoolData()`, and inform the spooler that the buffers are complete by calling `ssplProcessSignals()`:

```
int32 ssplProcessSignals (SoundSpooler *sspl, int32 SignalMask,  
    void (*UserBufferProcessor) (SoundSpooler *sspl,  
        SoundBufferNode *sbn))
```

You must also read more sound data and keep calling `ssplSpoolData()` until you run out of data.

6. Stop the sound spooler by calling `ssplStopSpooler()`:

```
Err ssplStopSpooler (SoundSpooler *sspl)
```

7. Clean up by the spooling process by calling `ssplDeleteSoundSpooler()`:

```
Err ssplDeleteSoundSpooler (SoundSpooler *sspl)
```

An Example

The following example uses parts of the sample program *ta_spool.c* (located in *Examples/Audio/Spooling*) to show how to use the sound spooler functions in your application program. Each step correlates to the steps given in "How to Use the Sound Spooler" on page 132.

**1. Load
sample
player**

```
/* Use line_out.dsp */
OutputIns = LoadInstrument("line_out.dsp", 0, 100);
CHECKRESULT(OutputIns, "LoadInstrument");
Result = StartInstrument( OutputIns, NULL );
CHECKRESULT(Result, "StartInstrument: OutputIns");

/* Load fixed rate stereo sample player instrument */
SamplerIns = LoadInstrument("sampler_16_f2.dsp", 0, 100);
CHECKRESULT(SamplerIns, "LoadInstrument");

/* Connect Sampler Instrument to Output Instrument */
Result = ConnectInstrumentParts (
    SamplerIns, "Output", AF_PART_LEFT,
    OutputIns, "Input", AF_PART_LEFT);
CHECKRESULT(Result, "ConnectInstruments");
Result = ConnectInstrumentParts (
    SamplerIns, "Output", AF_PART_RIGHT,
    OutputIns, "Input", AF_PART_RIGHT);
CHECKRESULT(Result, "ConnectInstruments");
```

**2. Create data
structure**

```
/* Create SoundSpooler data structure. */
sspl = ssplCreateSoundSpooler( NUM_BUFFERS, SamplerIns );
if( sspl == NULL )
{
    ERR("ssplCreateSoundSpooler failed!\n");
    goto cleanup;
}
```


3. Fill the queue

```

/* Fill the sound queue by queuing up all the buffers. "Preroll" */
BufIndex = 0;
SignalMask = 0;
for( i=0; i<NUM_BUFFERS; i++)
{
/* Dispatch buffers full of sound to spooler. Set User Data to
** BufIndex. ssplSpoolData returns a signal which can be checked
** to see when the data has completed it playback. If it returns 0,
** there were no buffers available.
*/
    Result = ssplSpoolData( sspl, Data[BufIndex], SAMPSIZE,
        NULL );
    CHECKRESULT (Result, "spool data");
    if (!Result)
    {
        ERR(("Out of buffers\n"));
        goto cleanup;
    }
    MySignal[BufIndex] = Result;
    SignalMask |= MySignal[BufIndex];
    BufIndex++;
    if(BufIndex >= NUM_BUFFERS) BufIndex = 0;
}

```

4. Start the sound spooler

```

/* Start Spooler instrument. Will begin playing any queued buffers. */
Result = ssplStartSpoolerTagsVA (sspl,
    AF_TAG_AMPLITUDE_FP, ConvertFP_TagData(1.0),
    TAG_END);
CHECKRESULT(Result, "ssplStartSpoolerTags");

```

**5. Monitor
the signals
and feed
data to the
spooler**

```
/* Play buffers loop. */
do
{
/* Wait for some buffer(s) to complete. */
CurSignals = WaitSignal( SignalMask );

/* Tell sound spooler that the buffer(s) have completed. */
Result = ssplProcessSignals( sspl, CurSignals, NULL );
CHECKRESULT(Result, "ssplProcessSignals");

/*
** Spool as many buffers as are available.
** ssplSpoolData will return positive signals as long as it
** accepted the data.
*/
while ((Result = ssplSpoolData (sspl, Data[BufIndex],
SAMPSSIZE, NULL)) > 0)
{

/* INSERT YOUR CODE HERE TO FILL UP THE NEXT BUFFER */

BufIndex++;
if(BufIndex >= NUM_BUFFERS) BufIndex = 0;
}
CHECKRESULT (Result, "spool data");
}
.
.
.
```

**6. Stop the
sound
spooler**

```
/* Stop Spooler. */
Result = ssplStopSpooler( sspl );
CHECKRESULT(Result, "StopSoundSpooler");
.
.
.
```

**7. Cleanup
the spooling
process**

```
ssplDeleteSoundSpooler( sspl );
UnloadInstrument( SamplerIns );
UnloadInstrument( OutputIns );
```

Convenience Functions

The sound spooler contains two convenience functions: `ssplSpoolData()` and `ssplPlayData()`.

ssplSpoolData()

`ssplSpoolData()` spools data by requesting a free buffer and then uses it if it is available. The function is listed below; see "ssplSpoolData" in the *3DO Music and Audio Programmer's Reference* for complete details.

```
int32 ssplSpoolData (SoundSpooler *sspl, char *Data,
                    int32 NumBytes, void *UserData)
{
    SoundBufferNode *sbn;
    int32 result = 0;

    /* Request a buffer, return if one isn't available */
    if ((sbn = ssplRequestBuffer (sspl)) == NULL) goto clean;

    /*
       Set address and length of buffer.
       Return buffer to spooler on failure.
    */
    if ((result = ssplSetBufferAddressLength (sspl, sbn, Data,
                                             NumBytes)) < 0) {

        ssplUnrequestBuffer (sspl, sbn);
        goto clean;
    }

    /* Set User Data (can't fail). */
    ssplSetUserData( sspl, sbn, UserData );

    /* Send that buffer off to be played. */
    return ssplSendBuffer (sspl, sbn);

clean:
    return result;
}
```

ssplPlayData()

ssplPlayData() spools data by requesting a free buffer. If no buffers are available, it waits for one and then uses it. The function is listed below; see "ssplPlayData" in the *3DO Music and Audio Programmer's Reference* for complete details.

```
int32 ssplPlayData (SoundSpooler *sspl, char *Data, int32 NumBytes)
{
    SoundBufferNode *sbn;
    int32 result;

    /*
     * Request a buffer. If no buffers are available,
     * wait for some buffer signals to arrive and process
     * them to free up a buffer.
     */
    while ((sbn = ssplRequestBuffer (sspl)) == NULL) {
        const int32 sigs = WaitSignal (sspl->sspl_SignalMask);

        if ((result = ssplProcessSignals (sspl, sigs, NULL)) < 0)
            goto clean;
    }

    /*
     * Set address and length of buffer.
     * Return buffer to spooler on failure.
     */
    if ((result = ssplSetBufferAddressLength (sspl, sbn, Data,
        NumBytes)) < 0) {

        ssplUnrequestBuffer (sspl, sbn);
        goto clean;
    }

    /* Send that buffer off to be played. */
    return ssplSendBuffer (sspl, sbn);

clean:
    return result;
}
```

Sound Spooler Function Calls

The sound spooler currently includes these function calls, grouped here by the type of function that each performs. The calls can be broken down into four different categories:

Convenience calls

The following two calls are convenience calls:

<code>ssplPlayData()</code>	Waits for next available buffer then sends a block full of data to the DSP.
<code>ssplSpoolData()</code>	Send a block full of data.

SoundSpooler management calls

The following calls are used to manage the sound spooler:

<code>ssplAbort()</code>	Abort SoundSpooler.
<code>ssplCreateSoundSpooler()</code>	Create a SoundSpooler data structure and initialize it.
<code>ssplDeleteSoundSpooler()</code>	Delete a SoundSpooler data structure.
<code>ssplGetSpoolerStatus()</code>	Get SoundSpooler status flags.
<code>ssplPause()</code>	Pauses the SoundSpooler.
<code>ssplProcessSignals()</code>	Process completion signals that have been received.
<code>ssplReset()</code>	Reset Sound Spooler.
<code>ssplResume()</code>	Resume SoundSpooler playback.
<code>ssplSetSoundBufferFunc()</code>	Install new SoundBufferFunc in SoundSpooler.
<code>ssplStartSpoolerTags()</code>	Start SoundSpooler.
<code>ssplStopSpooler()</code>	Stop SoundSpooler.

SoundBufferNode management calls

The following calls help manage a SoundBufferNode:

<code>ssplGetSequenceNum()</code>	Get sequence number of a SoundBufferNode.
<code>ssplGetUserData()</code>	Get user data from a SoundBufferNode.

<code>ssplRequestBuffer()</code>	Asks for an available buffer.
<code>ssplSendBuffer()</code>	Send a buffer full of data.
<code>ssplSetBufferAddressLength()</code>	Attach sample data to a <code>SoundBufferNode</code> .
<code>ssplSetUserData()</code>	Store user data in a <code>SoundBufferNode</code> .
<code>ssplUnrequestBuffer()</code>	Return an unused buffer to the sound spooler.

Low level calls

The following low level calls help manage the sound spooler:

<code>ssplAttachInstrument()</code>	Attach new sample player to instrument <code>SoundSpooler</code> .
<code>ssplDetachInstrument()</code>	Detach the current sample player instrument from the <code>SoundSpooler</code> .

Callback routine

The following call is used by `ssplProcessSignals()`:

<code>SoundBufferFunc</code>	<code>SoundSpooler</code> callback function typedef.
<code>UserBufferProcessor</code>	Callback function called by <code>ssplProcessSignals()</code> .

Playing MIDI Scores

This chapter discusses techniques that use the Music library to import a MIDI score from a standard MIDI file and then play the score back using Audio folio resources. The MIDI Score System can also be used as a high level sound effects manager.

This chapter shows how Music library functions can import a MIDI score file from an external source (a composition program on the Macintosh, for example) and turn that score into a Juggler object that can be played using Audio folio instruments, all without a MIDI port or any internal MIDI hardware. The chapter takes a close look at different aspects of MIDI score playback: the process of score translation, the MIDI environment created by the Music library, and the process of MIDI message playback using the Juggler.

To understand the topics discussed here, you need a good working knowledge of the MIDI standard, how it is usually applied to synthesizer setups, what constitutes a MIDI message, and the variety of messages available. There is a brief review of MIDI basics at the beginning of this chapter, but explaining the full MIDI standard is beyond the scope of this book.

If what you really want to do is simply set up a task to play back a MIDI score without having to know all the details behind playback, you can use the example program, *playmf.c*. The source code for this program is included on the Portfolio release disc. *playmf.c* is set up in modules that allow you to easily modify the code to play your MIDI score. You can refer back to sections of this chapter as necessary if you do not understand what is going on, and what needs to be changed. The example file *MFLoopTest.c* also provides an example of playing MIDI scores.

If you do not need to write MIDI playing code, and are only interested in creating a MIDI score for playback, then the two sections of this chapter you need to read are "Setting PIMap Entries" on page 160 and "Creating a MIDI Score for Playback" on page 180.

This chapter contains the following topics:

Topic	Page Number
A Brief MIDI Review	151
Creating an Internal MIDI Environment	153
An Overview of the MIDI Score-Playing Process	157
Setting Up	158
Creating a MIDI Environment	158
Importing a MIDI Score	165
Setting the Tempo	168
Playing the MIDI Score	168
Using MIDI Functions	172
Changing Characteristics During Playback	178
Cleaning Up	179
Creating a MIDI Score for Playback	180
Primary Data Structures	181
Function Calls	181

A Brief MIDI Review

This section provides a review of MIDI basics as they apply to the Music library.

MIDI Channels

Although MIDI was originally created to send musical events in live performance, it quickly evolved into a way to store timed musical events in a MIDI score. The simplest MIDI scores consist of a single sequence, an ordered list of consecutive MIDI messages that are sent out via MIDI cables to attached MIDI devices. Within that sequence, some MIDI messages can be transmitted to a specific *MIDI channel*.

MIDI channels (which range in number from 1 to 16) allow MIDI messages to be sent to specific devices within a MIDI network. Any connected MIDI device can be set to receive a specific channel; the device then responds only to MIDI messages specified for that channel. Other devices not set to that channel do not respond to those messages. For example, consider a synthesizer set to channel 5. It plays notes when it receives Note On messages for channel 5. It does not play notes when it receives Note On messages for channel 7 or any channels other than channel 5.

Today's synthesizers may combine the features of several polyphonic synthesizers into one device. As a result, one synthesizer can now listen to several MIDI channels simultaneously and devote many voices to each channel. The voices dedicated to a single channel all play using a single program, so they have the same timbre and sound like a single instrument. Each channel can be set to a different program, so the net result can be, on a single MIDI device, one instrument sound (a program) playing for each channel.

(The unfortunate MIDI term "program" seems designed to confuse, with different meanings for a programmer, who knows it as a piece of software, or for a classical musician, who knows it as a collection of musical pieces. In the world of MIDI, a program is a preprogrammed sound design, the equivalent of an instrument template in the Audio folio. A better word for program is "patch," but it is not commonly used in the world of MIDI, so "program" is used in this chapter.)

The Portfolio Music library sets up 3DO audio to act as a virtual MIDI device. This device is capable of receiving messages on many different MIDI channels, and can play using a different Audio folio instrument template (a different program) for the notes of each channel. It is useful to think of each channel specified within a sequence as a collection of notes to be played using a distinct instrument template. For example, channel 1 notes can be played using a sampled guitar template, while channel 2 notes can be played using a sawtooth wave template. The instrument template used by the channel can be changed at any time. For example, channel 1 can change from a sampled guitar template to a sampled piano template or to any other available instrument template.

Channel Messages

MIDI channel messages (that is, MIDI messages that can be assigned to a specific channel) describe many common musical events. The most common messages include:

- ◆ *Note On* turns on a voice in a receiving instrument and sets the pitch and amplitude of the voice. The pitch ranges from 0 to 127, as does the amplitude.
- ◆ *Note Off* turns off a voice at a set pitch in a receiving instrument. The pitch specified can range from 0 to 127, and a second value from 0 to 127 can optionally set the release speed of the note.
- ◆ *Program Change* specifies a new program (numbered from 0 to 127) for all receiving instruments tuned to the channel.
- ◆ *Control Messages* specify a change to a real-time parameter such as volume, panning, modulation, and so on. The score player presently only supports volume (controller 7), to set the overall volume of the channel, and pan (controller 10), to pan the stereo location of the channel playback from left to right.
- ◆ *Pitch Bend Change* specifies an amount of pitch bend (from 0 to 16,383) to apply to all voices in a channel. A value of 8192 specifies no pitch bend. Values above specify bend above the original pitch, values below specify bend below the original pitch.

There are many other types of MIDI messages, but the Music library is currently designed to respond only to the fundamental messages listed above. It ignores any other MIDI messages it receives during playback.

MIDI Score Playback

The MIDI standard defines several different MIDI score formats. The Music library can work with two of these formats:

- ◆ Format 0 contains a single sequence of MIDI events.
- ◆ Format 1 contains one or more sequences of MIDI events. Each sequence is called a *track*.

Within each track (sequence) contained in a score is a consecutive list of MIDI messages. Each track (sequence) can contain events for all 16 MIDI channels. Each message has a time value associated with it; the time is measured in relative units called *MIDI timing clocks*. MIDI traditionally defines a quarter note as equal to 24 clocks because 24 can be divided evenly by 2, 3, 4, 6, 8, and 12.

A MIDI score in playback is controlled by a series of MIDI timing clock messages sent to a sequencer that stores the score. The sequencer counts the MIDI clocks it receives and sends out the appropriate MIDI messages when it perceives the right amount of time has passed. For example, consider a sequence with a series of

quarter notes. The sequencer plays the first quarter note when it receives the first MIDI timing clock message. Twenty-four timing clocks later, it plays the next quarter note. Twenty-four timing clocks after that, it plays the next quarter note. And so on, until there are no more notes left to play.

Because timing clocks are a completely relative time unit, the rate of the timing clocks sent to the sequencer controls the tempo of the score playback. Send clocks at a faster rate, and the score plays faster. Send clocks at a slower rate, and the score plays slower.

Creating an Internal MIDI Environment

Because a basic 3DO unit has no MIDI ports or internal MIDI hardware, the Music library must import MIDI messages from a score file. It must then create an internal MIDI environment that uses the audio voices of a 3DO unit for note playback. This effectively turns the 3DO audio system into a multichannel polyphonic synthesizer. The system is set up to receive MIDI messages extracted from the file, act on them, and keep track of its own internal state as MIDI messages change parameters such as overall volume or number of voices currently being played.

To play MIDI messages in this internal MIDI environment, the Music library must be able to translate a score file's messages into events that the Juggler can read. The Juggler can then use appropriate software MIDI playback mechanisms for each event. Once the score is translated, the Music library must be able to control score timing, that is, the rate at which it feeds the score's MIDI messages to MIDI playback mechanisms. This is the same type of score timing used as a standard MIDI sequencer sets the tempo according to the number of MIDI clocks it receives.

The following sections describe the MIDI environment set up by the Music library, an environment of Juggler objects and data structures where the MIDI playback functions can go to work.

Creating a Virtual MIDI Device

To play an imported MIDI score, a task must first create a virtual MIDI device using Music library elements. The virtual MIDI device responds to MIDI messages and keeps track of its own state. While playing, a hardware MIDI device has an internal state that changes during playback. The number of voices playing can increase and decrease, the program assigned to each channel can change, the overall volume can go up or down, the stereo mix can pan left and right, and many other conditions can change.

The virtual MIDI device created by the Music library also changes its state during playback. To keep track of the device's current state, the Music library provides a set of data structures designed to store current setting values. The values within

these data structures are maintained internally by the Music library, so a task need not (and *should not*) write to them directly. Nevertheless, it is useful to know what these data structures are and how they are used because a task playing back a MIDI score needs to provide these data structures for score playback.

Providing Voices

A dedicated synthesizer usually has a fixed set of voices. Each voice provides the capability to play a single note at one time. The more voices the synthesizer has, the more notes it can play at once. For example, a 16-voice synthesizer can play up to 16 notes at one time. If the synthesizer is a multichannel synthesizer, it can allocate those voices to different channels. Each channel can then use its allocated voices to play the program assigned to the channel. For example, if a 16-voice synthesizer assigns nine voices to channel 1 and seven voices to channel 2, then channel 1 can play up to nine simultaneous notes, while channel 2 can play up to seven simultaneous notes.

When a hardware synthesizer plays notes, it uses one voice for the first note, and then, if a second note begins before the first finishes, it plays the second note on a second voice so the notes sound simultaneously. Whenever a new note starts, it uses an unused voice, unless there are no unused voices available, in which case it steals a used voice to start the new note.

To provide the equivalent of a voice in a dedicated synthesizer, the Music library uses the `NoteTracker` data structure, one for each voice. Each `NoteTracker` data structure is tied to an audio instrument and keeps track of whether the instrument is playing a note or not. If it is playing a note, and score playback requires another note to be started, the Music library's voice allocation looks for a free `NoteTracker`. It then plays the note using that `NoteTracker` voice and marks the `NoteTracker` data structure to show that a note is playing.

Note that the `NoteTracker` data structure is only used internally by Music library functions. A task should not write values directly to a `NoteTracker` data structure, just as a task should not touch other internally used data structures.

Setting Channels

The virtual MIDI device, like a dedicated synthesizer, must keep track of the state of its channels. To do so, it uses the `ScoreChannel` data structure, which holds values that, among other things, give the default program for the channel, the current program setting, the priority of the channel's voices, the number of voices assigned to the channel, the amount of pitch bend set for the channel, and the overall volume of the channel. The `ScoreChannel` data structure, like the `NoteTracker` data structure, is for Music library internal use only and a task should not write to it directly.

Assigning Instrument Templates to Program Numbers

MIDI messages use programs and program numbers where the Audio folio uses instrument templates and template names. To translate, the Music library uses a PI (program-to-instrument) map to associate program numbers with instrument template names. The PIMap is an indexed array of instrument template names. When a MIDI message supplies a program number, the function accepting the number uses it as an index into the PIMap. It retrieves the appropriate instrument template name and uses that template as the timbre for the specified program.

Keeping Track of Overall Playback

To keep track of overall playback, the Music library supplies the `ScoreContext` data structure. This data structure contains, among other things, a pointer to the PIMap structure used for score playback, the maximum number of voices available (to limit strain on system resources), the maximum volume allowed per voice, the name of a mixer instrument used to accept the output of `NoteTracker` voices, the right and left gain values for the mixer, a pointer to an array of `NoteTracker` data structures used for playback, a list of free `NoteTrackers` available to play a note, and an array of `ScoreChannel` data structures, one for each possible channel. The `ScoreContext` data structure sets the score context, the overall control for this virtual MIDI device.

The `ScoreContext` data structure is another Music library internal data structure that should not be written to directly by a task.

Importing a MIDI Score

The Music library provides two functions that import and translate MIDI scores: `MFLoadCollection()`, which handles MIDI format 1 scores, and `MFLoadSequence()`, which handles MIDI format 0 scores. Both of these functions are described in “Importing a MIDI Score” on page 165.

When `MFLoadSequence()` imports a Format 0 score, it turns the score into a single Juggler sequence. When `MFLoadCollection()` imports a Format 1 score, it turns the score into a Juggler collection that contains a Juggler sequence for each of the MIDI score's component tracks. Each track within a MIDI score, whether Format 1 or Format 0, is always translated into a single Juggler sequence, whether the sequence stands alone (for a Format 0 score) or is part of an encompassing collection (for a Format 1 score).

Within a Juggler sequence created from a MIDI score, each MIDI message is turned into a `MIDIEvent` data structure, a form that the Juggler can handle directly during playback. The `MIDIEvent` data structure is defined as follows:

```
typedef struct MIDIEvent
{
    uint32    mev_Time;
    unsigned char mev_Command;
    unsigned char mev_Data1;
    unsigned char mev_Data2;
    unsigned char mev_Data3;
} MIDIEvent;
```

The first element of the structure is the MIDI message's time clock value, which is translated here into a 32-bit unsigned value that gives an equivalent number of audio ticks. For example, a MIDI message with a timing clock value of 328 is assigned an audio tick value of 328. Audio tick values are ticked off by the audio clock to time events during playback by the Juggler.

The second element of the structure is the single-byte MIDI message type. The third, fourth, and fifth elements are single-byte data values that may (or may not) accompany a MIDI message. If a MIDI message type does not require accompanying data or does not require all three bytes of data, the contents of the unused data elements are not relevant and are not read.

Providing MIDI Playback Functions

Once a MIDI score is in place as a Juggler object, and a MIDI environment is set up using the proper data structures, the Juggler plays the object and the MIDI messages stored in the object's events must be properly executed. To do so, the Music library provides a collection of MIDI playback functions that act according to the MIDI messages in the score. These functions initiate Audio folio activities to carry out the action called for by a MIDI message. They also update the appropriate data structures to reflect their actions.

The MIDI Interpreter Procedure

Every Juggler sequence includes a pointer to an interpreter procedure. As the events in the sequence are played, the interpreter procedure receives the data associated with each event and processes that data. When a MIDI score is imported as a Juggler object, that Juggler object points to the Music library function `InterpretMIDIEvent()` as its interpreter procedure. When the

Juggler plays the MIDI score object, its events are passed to `InterpretMIDIEvent()`, which strips out the MIDI message embedded in the event and passes it on to `InterpretMIDIMessage()`.

`InterpretMIDIMessage()` (discussed in “The Interpreter Procedure” on page 167) reads the MIDI message and calls an appropriate Music library function to act on that message type. `InterpretMIDIMessage()` passes along the data that accompanies the MIDI message. The function that acts on the MIDI message, makes the appropriate Audio folio calls, and updates the appropriate Music library data structures.

Setting the Audio Clock Speed to Control Tempo

An imported MIDI score has no innate tempo, only arbitrary time values measured in audio ticks. Because a MIDI sequence is played back using a `BumpJuggler()` loop (a technique described in Chapter 10, “Creating and Playing Juggler Objects”), the audio clock controls the playback tempo. A MIDI score is usually saved with a particular tempo measured in MIDI clocks per second. The audio clock should be set to the same tempo, measuring audio ticks per second instead of MIDI clocks per second. To do so, the task playing back the score creates a custom audio clock and sets its rate as described in Chapter 6, “Advanced Audio Folio Usage.” Once the clock rate matches the prescribed tempo, the MIDI environment is set for playback.

An Overview of the MIDI Score-Playing Process

This section shows how a task uses the components of the MIDI environment to play back a MIDI score. A task typically follows these steps:

1. Open the Audio folio and initialize the Juggler. This opens the Audio folio so that the Music library can use instruments and the audio clock. It also sets up the Juggler to handle MIDI playback.
2. Create a MIDI environment.
 - ◆ Set the maximum number of voices and programs available. This limits the number of instrument templates and DSP resources required.
 - ◆ Create a `ScoreContext` data structure. This step also includes creating attendant data structures such as `ScoreChannel` data structures.
 - ◆ Create a `PIMap`. This step imports the necessary instrument templates and then associates them with corresponding program numbers.
 - ◆ Create a mixer for connecting voice output to the DAC. This step also creates the `NoteTracker` data structures necessary for allocating voices, and allocates system amplitude for the mixer.

3. Import a MIDI score file.
 - ◆ Import a MIDI file from CD-ROM or other source. This translates MIDI messages and places them in a juggler sequence or collection. It also sets up an interpreter procedure for the juggler object that correctly interprets the MIDI messages in the object.
 - ◆ Specify a score context for the imported score. This sets a pointer in the juggler object to the score context necessary for playback.
4. Adjust score timing to match the MIDI song tempo. This step either sets the audio clock to an appropriate rate, or it scales MIDI timestamps for each score event.
5. Play the score in a `BumpJuggler()` loop.
6. Clean up.
 - ◆ Free allocated amplitude.
 - ◆ Reset the audio clock.
 - ◆ Delete all the instrument templates.
 - ◆ Delete the score context and attendant data structures.
 - ◆ Terminate the juggler.
 - ◆ Close the Audio folio.

The following sections discuss the details for each of these steps.

Setting Up

Before a task can play a MIDI score, it must first set up by opening the Audio folio with the `OpenAudioFolio()` call. It must also initialize the juggler with the `InitJuggler()` call.

Creating a MIDI Environment

Creating a MIDI environment is something like designing a synthesizer. A task defines how many voices it uses, how many programs it supports, and what instrument templates it uses for different program numbers. It then sets up data structures to manage its voices and programs.

Setting Voice and Program Limits

When a task is set up to play a MIDI score, it should put upper limits on the number of voices it needs and the number of programs it responds to. These limits should be as low as possible to save system resources.

Each program supported requires an instrument template, which can take up quite a bit of RAM, especially if it is a sampled sound instrument template. Therefore, the fewer programs supported, the better. To reduce the memory

required for the PIMap, the original MIDI score should be created to use the lowest possible program number for its program changes. For example, if a score uses nine programs, they should have program numbers 0 to 8, not numbers scattered throughout the range of 0 to 127. The original score should not use program numbers that are higher than the maximum number of programs, because there is no PIMap entry to support a number that high.

Each voice supported requires DSP resources to play a note, a NoteTracker data structure to keep track of its status, and a mixer input. So reducing the number of voices supported is important to conserve system resources. The most likely limiting factor is DSP resources, which are used up by the number and type of instruments playing. If all instruments are simple and use few DSP resources, then the DSP can support many voices. If instruments are complex, use many DSP resources, and many of them play at once, then the DSP cannot support as many voices.

A task should support no more than the maximum number of voices required by the original MIDI score. The original score should be created with careful consideration of the type of instruments used to play its notes. If the instruments are complex, the score may have to use fewer voices so that it does not overtax DSP resources.

A task can define its maximum voice and program support in constants that are used in later Music library calls. For example,

```
#define MAX_NUM_PROGRAMS (9)
#define MAX_NUM_VOICES (8)
```

These values can be used later to set program and voice limits. The call `CreateScoreContext()` sets the maximum number of programs, and the call `InitScoreMixer()` or `InitScoreDynamics()` sets the maximum number of voices.

Creating a Score Context

After you have decided on limits for program and voice support, you can proceed to create a score context using this call:

```
ScoreContext *CreateScoreContext( int32 MaxNumPrograms )
```

The call accepts a single argument, `MaxNumPrograms`, which sets the number of programs supported by the score context. This number can be no greater than 128. If the task defined this limit earlier with a constant, it can supply that constant as the argument.

When `CreateScoreContext()` executes, it creates a `ScoreContext` data structure and sets its elements to default settings. That data structure includes an array of `ScoreChannel` data structures to control channels.

`CreateScoreContext()` also creates a PIMap array with as many elements as

it needs to accommodate the number of programs specified. The `ScoreContext` data structure includes a pointer to that array. It also includes a pointer to an array of `NoteTrackers`, which is set later when the `NoteTrackers` are created with the `InitScoreDynamics()` call.

`CreateScoreContext()` returns a pointer to the created `ScoreContext` data structure if successful. If unsuccessful, it returns a negative value (an error code).

Setting PIMap Entries

After you have created a score context and the `PIMap` that goes with it, you must set the entries in the `PIMap` to show which instrument template is designated for each program number. You can do this in one of two ways: you can first load the instrument templates you need and then directly set each of the `PIMap`'s entries using the `SetPIMapEntry()` call. Or you can specify instrument templates and program numbers in a text file and use the `LoadPIMap()` call to read the file, load the appropriate instrument templates, and make the appropriate `PIMap` entries. The second method is the simpler of the two.

Directly Setting PIMap Entries

If you want to directly set a score context's `PIMap` entries, you must first load the instrument templates you want to specify in the entries and store the item number for each loaded template. You then use this call:

```
int32 SetPIMapEntry( ScoreContext *scon, int32 ProgramNum,
                    Item InsTemplate, int32 MaxVoices, int32 Priority )
```

The call accepts five arguments: The first, `*scon`, is a pointer to the score context that owns the `PIMap`. The second, `ProgramNum`, is a value from 0 to 127 that specifies which `PIMap` entry to set. The third, `InsTemplate`, is the item number of an instrument template to be assigned to the specified `PIMap` entry. The fourth is the maximum number of voices that can be played simultaneously with this program. And the fifth is the priority number, from 0 to 200, for this program.

There are some important considerations for these arguments. First, you should note that program numbers in a MIDI message range from 1 to 127. Most synthesizer manufacturers consider that odd, so they add one (1) when displaying program numbers, causing the range to go from 1 to 128. In general, user interfaces that are designed for musicians rather than programmers number programs from 1 to 128. User interfaces designed for programmers instead of musicians number programs from 0 to 127. When you set the `ProgramNum` argument for this call, keep this discrepancy in mind if you are trying to match the program number in the original score.

You should also note that the maximum number of voices set by `MaxVoices` is *not* the maximum number of voices available to the full score context. It is the maximum number of voices for that program alone. It should be a number equal

to or less than the maximum number of voices available to the score. This value allows you to limit the number of voices played using a complex instrument template that takes up DSP resources. For example, you may wish to limit a program using a complex lead instrument to two voices so that it does not overload the DSP by playing too many notes simultaneously.

The `Priority` argument sets a program to use a low-priority instrument (set with a lower value) or a high-priority instrument (using a higher value). When voices are allocated to notes, the Music library is more likely to steal a voice from a low priority program than it is to steal a voice from a high priority program. Set this argument to a low value if the program is not likely to be used for conspicuous notes, for example, a chorus instrument. Set it to a high value if the instrument is used for solo work. Set all `PIMap` entries to the same priority (100, for example) if you do not want any program to have priority over another.

When `SetPIMapEntry()` executes, it finds the `PIMap` associated with the specified score context, finds the specified entry, and then writes the instrument template item number, maximum voice value, and priority value to that entry. It returns 0 if successful, or a negative value (an error code) if unsuccessful.

Creating a PIMap File

A simpler way to set up a `PIMap` is to define it in a text file (strict ASCII) using the `PIMap` file format. You can then ask `LoadPIMap()` to read the file, load the appropriate instrument templates, and set the appropriate `PIMap` entries. The file format for a `PIMap` file is simple:

- ◆ A file consists of a line of text for each program-to-instrument description. Each line is separated from the next by a carriage return (0x0D).
- ◆ Each line starts with a MIDI program number in the range 1 to 128.
- ◆ The program number is followed by the name of an instrument template or the name of a sample to be played by an instrument.
- ◆ The name is followed by options and option values if desired. These are the options currently available:
 - ◆ -f, which follows a sample name and specifies that the sample's pitch remains fixed and not variable, a useful feature for percussion instruments. Fixed pitch instruments are also more efficient and do not use as many DSP resources.
 - ◆ -d, which detunes an instrument by a specified number of cents. It is followed by a cent value, which can range from -50 to +50.
 - ◆ -m, which is the maximum number of voices that can play for this program. The default is 1.
 - ◆ -l, which sets the low note for a sample. It is followed by a MIDI pitch value from 0 to 127.

- ◆ -b, which sets the base note for a sample. It is followed by a MIDI pitch value from 0 to 127.
- ◆ -h, which sets the high note for a sample. It is followed by a MIDI pitch value from 0 to 127.
- ◆ -p, which sets the priority for an instrument template. It is followed by a value from 0 to 200. The default is 100.
- ◆ -r N, which allows one to specify the rate at which a DSP instrument executes. N can be 1 or 2. Default is 1. If N is 2, the execution rate is divided by 2, and the instrument only uses half the number of DSP ticks. This is useful if you are using an instrument that consumes more ticks than code words, such as `sampler_16_v1.dsp`. The sample rate is cut in half so the maximum pitch is reduced by a factor of two. The instrument also plays with less fidelity since the output is not interpolated up to 44 KHz. See the description of the `AF_TAG_CALCULATE_DIVIDE` tag in "Instrument" of the *3DO Music and Audio Programmer's Reference*.
- ◆ Elements within a line are separated by one or more spaces. *Do not* use tabs.
- ◆ A comment line starts with a semicolon (;) in the first column.

Each line of a PIMap file describes a single PIMap entry, starting with the program number, followed by an instrument template name or sample name to be used for that program number. The optional flags prescribe changes for the instrument template when used.

The Music library in general cleaves to the programmer's model of numbering programs from 0 to 127; the one exception is the PIMap file, which numbers programs from 1 to 128.

Setting Up Multisamples in a PIMap File

One important feature of a PIMap file is that you do not have to use the name of sampled sound instrument for a PIMap entry, you can use the name of the sample you wish to play instead. If the PIMap entry already has a sampled sound instrument template assigned to it, then the sample you specify for that entry is attached to that template. If the PIMap entry does not have a sampled sound instrument template assigned to it, then `LoadPIMap()` finds and loads the sampled sound instrument most appropriate for the sample.

Note that if you specify several different samples for the same PIMap entry, they are all attached to the sampled sound instrument template for that entry. This allows you to specify multisamples in the PIMap file for more realistic sounding sampled instruments or for varied timbres in one instrument, such as a drum kit. To create a multisample, simply list all of the samples required, one sample per line, and assign each sample to the same program number. The high and low ranges in the AIFF sample file can set the ranges of each sample in the multisample, or you can use the -l, -b, -h, and -d options to override those ranges and set new ones.

The instrument first specified for a program number is the instrument used to play all samples later assigned to that program number, so it is important to make sure that later samples match the program's instrument. If they do not match, chaos can follow. For example, if you first specify a 16-bit sample for program 4, `LoadPIMap()` loads a 16-bit sampled-sound instrument to play that sample. If you then assign an 8-bit sample to program 4, the 8-bit sample is played using the 16-bit sampled-sound instrument, a guaranteed muddle.

As a PIMap file example, consider the text file shown in Example 11-1. Notice several things: the gong is fixed in pitch; the clarinet is set up as a multisample with octave spacing; the organ has a maximum voice option set to four, which allows it to play chords of up to four voices; and the sample *bell.aiff* is guaranteed to be played by the instrument *sampler_16_v1.dsp* because the instrument is specified before the sample is specified. This technique is often used with ARIA instruments.

Example 11-1 *PIMap file example.*

```
; comments are allowed after a semicolon
; define multisample for clarinet
1 clarinet1.aiff -l 30 -b 48 -h 53
1 clarinet2.aiff -l 54 -b 60 -h 66
1 clarinet3.aiff -l 67 -b 72 -h 84
2 gong.aiff -f
3 bass.aiff -p 150
4 organ.aiff -m 4
; play bell using explicit instrument
5 sampler_16_v1.dsp
5 bell.aiff
```

Loading a PIMap File

After you have created a PIMap file, you can load it using this call:

```
int32 LoadPIMap( ScoreContext *scon, char *Filename )
```

It accepts two arguments: **scon*, a pointer to the score context whose PIMap you want to reset; and **Filename*, a pointer to a character string containing the filename of the PIMap file.

When it executes, `LoadPIMap()` reads the PIMap file and imports the specified instrument templates and sampled sounds. If an entry in the file specifies a sampled sound (it ends in *.AIF*, *.AIFF*, or *.AIFC*), `LoadPIMap()` attaches the sampled sound to a sampled sound instrument template if the entry already has one assigned. If the entry does not have a template assigned, `LoadPIMap()`

determines the appropriate sampled sound instrument to play the sampled sound, assigns it to the entry, and attaches the sampled sound. The call then revises all effected PIMap entries in the score context's PIMap.

`LoadPIMap ()` returns 0 if successful, or a negative value (an error code) if unsuccessful.

The highest program number in your PIMap file should not exceed the number of entries in the score context's PIMap.

Setting Up a Mixer

After instrument templates have been loaded and the PIMap is set, you should load an appropriate mixer instrument template and create a mixer instrument. This mixer handles the note output from voices as they play a MIDI score, combining them and sending them through stereo outputs to the digital-to-analog converter (DAC). As the MIDI playback functions play notes, they create new instruments, when appropriate, to play the notes. Each new instrument is connected to a mixer input, where the instrument can be panned from left to right in the mixer's stereo output. (The panning, initially set to center, is under the control of MIDI pan control messages.) When an instrument is freed, it is detached from the mixer so the mixer output is available for another instrument.

To create a mixer for use with a Score, call `CreateScoreMixer()`.

```
Err CreateScoreMixer( ScoreContext *scon, MixerSpec mixerSpec,  
    int32 maxNumVoices, float32 amplitude );
```

The parameter `maxNumVoices` determines the maximum number of voices that can be allocated and tracked by the dynamic voice allocator. This number should not exceed the number of inputs on the mixer. The amplitude parameter is the maximum gain on the mixer for a sound that is panned hard to the left or right. The `MixerSpec` describes the mixer that is to be used.

Setting an Amplitude Value

When you supply an amplitude value for `MakeMixerSpec ()`, you can be assured that you never overload the DAC if you take the maximum available system amplitude, divide it by the maximum number of voices used, and take the resulting value as the maximum voice amplitude. This means that if all voices sound simultaneously at maximum amplitude and all panned to one side, the combined signal will not exceed the maximum system amplitude. In practice, it is an extremely rare occurrence for this to occur, so you may want to increase the maximum voice amplitude by 200 or 300 percent to give the resulting audio signal a boost. There is a slim chance that you may clip the signal, but in most cases it is highly unlikely. If you hear clipping, reduce the amplitude.

Importing a MIDI Score

After a MIDI environment is prepared with a score context and its attendant data structures, a task can import a MIDI score file for playback there. It helps if that MIDI score is tweaked to best fit into the 3DO MIDI environment. It should limit unnecessary programs and voices. It should plan, in its PIMap, to use less complex DSP instruments. The MIDI score should be stored in a Format 0 or Format 1 file, which is often available through a sequencer's or composition tool's Export command.

Declaring a MIDIFileParser Data Structure

Before a task can import a MIDI file, it must first declare a `MIDIFileParser` data structure. This structure keeps track of the overall score parameters as the MIDI data is translated into a Juggler object. These parameters include values such as the clocks-per-second tempo set in the MIDI score. The `MIDIFileParser` data structure is defined in the `midifile.h` header file, and should never be written to directly by a task, the MIDI loading calls use it for their own internal purposes.

Creating a Juggler Object

The next step in importing a MIDI score is to create a Juggler object appropriate for the imported score. If the score is a Format 0 file, you must create a single Juggler sequence to contain the translated score. If the score is a Format 1 file, you must create a Juggler collection that holds one Juggler sequence for each sequence in the score. (The translating call itself creates the Juggler sequences within the collection.)

Loading the Score

With a `MIDIFileParser` data structure and a Juggler sequence in place, you can load a Format 0 MIDI score file using this call:

```
int32 MFLoadSequence( MIDIFileParser *mfpptr,
                      char *filename, Sequence *SeqPtr )
```

The call takes three arguments: `*mfpptr`, a pointer to the `MIDIFileParser` data structure used for translating the MIDI score; `*filename`, a pointer to a character string containing the name of the MIDI score file; and `*SeqPtr`, a pointer to the Juggler sequence where the translated MIDI score should be stored.

When `MFLoadSequence()` executes, it opens the MIDI score file, translates its contents using the `MIDIFileParser` data structure, and stores the translated MIDI messages in the specified sequence as MIDI events. It writes information about the MIDI score in the `MIDIFileParser` data structure. If successful, `MFLoadSequence()` returns 0; if unsuccessful, it returns a negative value (an error code).

To load a Format 1 MIDI score file, you must have a Juggler collection in place (instead of a Juggler sequence used for a Format 0 score). The collection should be empty. Use this call to load the file:

```
int32 MFLoadCollection( MIDIFileParser *mfpptr,
    char *filename, Collection *ColPtr )
```

The call takes three arguments: **mfpptr*, a pointer to the *MIDIFileParser* data structure used for translating the MIDI score; **filename*, a pointer to a character string containing the name of the MIDI score file; and **ColPtr*, a pointer to the Juggler collection where the translated MIDI score should be stored.

When *MFLoadCollection()* executes, it opens the MIDI score file and translates its contents using the *MIDIFileParser* data structure. It creates a Juggler sequence for each track in the MIDI score, and stores the translated contents of each track in a separate Juggler sequence. *MFLoadCollection()* writes information about the MIDI score in the *MIDIFileParser* data structure.

Note that if *MFLoadCollection()* processes a Format 0 MIDI file, it treats it as a single-track Format 1 MIDI file and turns it into a single-sequence collection. This collection requires more processing overhead than the same score loaded by *MFLoadSequence()* as a Juggler sequence. If your MIDI score has only one channel and you can save it as a Format 0 file, then you should always load it as a sequence to save processing overhead.

If successful, *MFLoadCollection()* returns 0; if unsuccessful, it returns a negative value (an error code).

If you use data streaming to import the image of a MIDI score file, then use this call to turn the file image into a MIDI collection that can be played by the Juggler:

```
int32 MFDefineCollection( MIDIFileParser *mfpptr,
    char *Image, int32 NumBytes, Collection *ColPtr)
```

The call takes four arguments: **mfpptr*, a pointer to the *MIDIFileParser* data structure used for translating the MIDI score; **Image*, a pointer to the file image (which should be a string of bytes); *NumBytes*, the size of the MIDI file image in bytes; and **ColPtr*, a pointer to the Juggler collection where the translated MIDI score should be stored.

When *MFDefineCollection()* executes, it turns the MIDI file image into a collection just as *MFLoadCollection()* turns a MIDI file into a collection. *MFDefineCollection()* returns 0 if successful, or a negative value (an error code) if unsuccessful.

Specifying the User Context

Juggler objects can contain a pointer to a user context. This feature is designed specifically for MIDI playback. Once a MIDI score is translated and placed in a Juggler sequence or collection, that sequence or collection must be set with a

pointer to the score context used to play back the MIDI score. To do so, you use tag arguments and the `SetObjectInfo()` call as described in Chapter 10, "Creating and Playing Juggler Objects".

The code fragment in Example 11-2 shows how a collection containing an imported MIDI score is set to point to the score context. `CollectionPtr` is a pointer to the collection; `SconPtr` is a pointer to the score context.

Example 11-2 *Pointing to a score context.*

```
/* Define TagList */
Tags[0].ta_Tag = JGLR_TAG_CONTEXT;
Tags[0].ta_Arg = (void *) SconPtr;
Tags[1].ta_Tag = TAG_END;

SetObjectInfo(CollectionPtr, Tags);
```

After the pointer to the score context is set, the object's interpreter procedure can pass the pointer along to other MIDI calls as they execute the MIDI events in the object.

The Interpreter Procedure

When `MFLoadSequence()` and `MFLoadCollection()` turn a MIDI score into a juggler object, all translated sequences contain a pointer to an interpreter procedure. This call is the MIDI interpreter procedure:

```
int32 InterpretMIDIEvent( Sequence *SeqPtr,
                        MIDIEvent *MEvCur, ScoreContext *scon )
```

The call accepts three arguments: `*SeqPtr`, a pointer to the sequence for which it interprets; `*MEvCur`, a pointer to the current event in the sequence; and `*scon`, a pointer to the score context used to play the event.

When executed, `InterpretMIDIEvent()` extracts a MIDI message from the current event. The extracted message is string of one-byte values that can be from one to four bytes long. `InterpretMIDIEvent()` then calls `InterpretMIDIMessage()` and passes the MIDI message and score context pointer along to it.

`InterpretMIDIMessage()` looks at the MIDI message to see if it is one of the messages recognized by the Music library. If it is, `InterpretMIDIMessage()` calls whichever of these MIDI playback functions is appropriate for carrying out the message: `StartScoreNote()` for Note On messages, `ReleaseScoreNote()` for Note Off messages, `ChangeScoreControl()` for volume or panning messages, `ChangeScoreProgram()` for program change

messages, or `ChangeScorePitchBend()` for pitch bend change messages. Although these functions are called automatically during score playback, they can be used directly by a task, and are described more fully later in this chapter.

Setting the Tempo

Before using the Juggler to play back a MIDI object, you should set the playback tempo to match the imported MIDI score's original tempo.

Setting the Audio Clock Rate

To set the playback tempo by setting the audio clock rate, your task should first create a custom audio clock. You should then set the clock rate to match the score's MIDI clock rate. The task can change the clock rate at any time during playback to change the playback tempo of the score.

The functions for handling the audio clock are discussed in the "Advanced Audio Folio Usage" Chapter. The one piece of new information you need is where to get the score's MIDI clock rate. It is stored in the `MIDIFileParser` data structure used to import the MIDI score, in the `mfp_Rate` element after executing `MFLoadCollection()`.

The code fragment in Example 11-3 shows how a task resets the audio clock rate. `MFPParser` is a pointer to the `MIDIFileParser` data structure used to import the MIDI score.

Example 11-3 *Resetting the audio clock rate.*

```
ScoreClock = CreateAudioClock(NULL);  
SetAudioClockRate (ScoreClock, convertF16_FP(MFPParser.mfp_Rate));
```

Note that the audio clock is often used by other tasks such as audio and video streaming tasks, so changing the audio clock rate can sometimes wreak great havoc on those other tasks. Be sure that your task gets ownership of the clock before it changes the rate. That way, if it screws up other tasks, your task did the right thing by inquiring about ownership first.

Playing the MIDI Score

With the MIDI environment prepared, the MIDI score imported, and the audio clock rate set, you can then use the Juggler to play back the MIDI score. The playback technique is no different than it is for any Juggler object, a process described in the chapter Chapter 10, "Creating and Playing Juggler Objects." In

fact, if you have already created a standard function that plays a Juggler object using the audio clock, you can probably use it to play a MIDI object with little adaptation.

The process is as follows:

- ◆ Create an audio cue for audio clock notification.
- ◆ Poll the audio clock for the current time.
- ◆ Use the current time to start all objects to be played (in this case, the MIDI sequence or collection).
- ◆ Use the current time to bump the Juggler.
- ◆ Retrieve the next event time returned by the Juggler.
- ◆ Wait until the next event time.
- ◆ Repeat the last three steps until there are no more events to execute.
- ◆ Delete the audio cue.

The code fragment in Example 11-4 plays back a MIDI collection:

Example 11-4 *Playing a MIDI collection.*

```
MyCue = CreateItem ( MKNODEID(AUDIONODE,AUDIO_CUE_NODE), NULL );
CueSignal = GetCueSignal ( MyCue );

ScoreClock = CreateAudioClock (NULL);
/* Drive Juggler using audio clock. */
/* Delay start by adding ca. 1/2 second to avoid stutter on
startup. */
ReadAudioClock (ScoreClock & NextTime);

/* Start playback loop. */
NextTime + = 120;
StartObject ( JglPtr , NextTime, NumReps, NULL );
do
{
    /* Request a timer wake up at the next event time. */
    Result = SleepUntilAudioTime( ScoreClock, MyCue,
NextTime );
    CurTime = NextTime;

    /* Tell Juggler to do its thing. Result > 0 if done. */
    Result = BumpJuggler( CurTime, &NextTime, SignalsGot,
&NextSignals );

} while (Result == 0);

DeleteCue( MyCue );
/* Delete AudioClock(ScoreClock);
```

Notice that the fragment includes a delay of approximately 1/2 second, which is put in place to avoid a stutter of voices at the beginning of the score playback. Without the delay, the object is started at the same time that the Juggler is bumped, which can cause voice stuttering.

Dynamic Voice Allocation

As the Juggler plays back a MIDI object, the interpreter procedure turns the MIDI events into MIDI messages. The MIDI messages are passed on to `InterpretMIDIMessage()`, which is the beginning of a tree of MIDI playback functions designed to act appropriately on each recognizable MIDI message. As these functions play notes, they use a system of dynamic voice allocation, controlled by the calls and the MIDI environment's data structures (such as `NoteTracker`). This system provides multiple voices in different channels

playing different programs. Although the task playing the MIDI score cannot control this process directly, it is useful to know something about how it works so that MIDI scores can be tweaked to work to their best advantage.

When the first note of a score is played by a voice, a MIDI playback function creates an instrument to play that note and connects the instrument's output to the MIDI mixer. The voice's channel determines which instrument template should be used to create the instrument. When the note is released, its instrument is abandoned, that is, it remains in existence, connected to the mixer, but it no longer plays.

When any subsequent note in a score is played, a MIDI playback function looks for an abandoned instrument of the right template for the note. If it finds one, it uses that instrument to play the note. If it does not find one, it looks for an abandoned instrument of another template, frees that instrument, and creates a new instrument using the right template for the note. And if it does not find any abandoned instruments, it creates a new instrument for the note. When the note is finished, its instrument is abandoned so that other notes can use it.

New instrument allocation continues until the number of existing instruments hits the upper limit set in the score context or until the DSP runs out of resources. When that happens, and there are no abandoned instruments to use, a MIDI playback function starting a new note must steal an existing instrument from another note. To do so, it looks through all the channels to find the least prominent voice to steal. It looks for notes of lower priority, released notes that are dying away, or old notes that the listener cannot hear anymore. It then steals its voice, cutting off the note. The function *does not* steal a voice from any note of a higher priority than the starting note, if the high-priority note is still playing.

To sum up, a MIDI playback function playing a note uses this priority list to look for an instrument:

1. Adopt an abandoned instrument.
2. Create a new instrument.
3. Steal an instrument of lower priority, an instrument used by a released note, or an instrument used by an old note.

Freeing Instruments Created by Voice Allocation

Note that the process of dynamic voice allocation creates instruments as necessary, but never reduces the number of instruments. It recycles existing instruments when possible. This means that if dynamic voice allocation must create a large number of instruments for a part of a score with a large number of multiple voices, the created instruments remain in existence no matter how brief the multiple voice section of the score was. These unused instruments can tie up DSP resources.

To free instruments allocated by dynamic voice allocation, use this call:

```
Err PurgeScoreInstrument( ScoreContext *scon, uint8
                        Priority, int32 MaxLevel )
```

This call accepts three arguments: the first, **scon*, is a pointer to a *ScoreContext* data structure controlling playback; the second, *Priority*, is the maximum instrument priority to purge, for instruments that are still playing (in the range of 0 to 200). Instruments of higher priority than this may be purged if they have stopped playing; and the third, *MaxLevel*, is the maximum activity to purge (i.e. *AF_ABANDONED*, *AF_STOPPED*, *AF_RELEASED*, *AF_STARTED*).

This procedure returns a positive value if an instrument was actually purged, zero if no instrument matching the specifications could be purged, or a negative error code on failure.

Using MIDI Functions

When the Juggler plays a MIDI score, it automatically calls MIDI playback functions through the MIDI object's interpreter procedure. Those MIDI playback functions are also available directly to a task, which allows a task to feed MIDI messages to the functions using its own techniques. It also allows a task to perform MIDI actions such as starting and stopping a note, all without directly using MIDI messages. And because these MIDI functions use the MIDI environment's voice allocation mechanism, their effects are fully integrated with a score that might be playing in the background or other audio activities taking place in the MIDI environment.

This section explains how the MIDI functions work so that a task can use them directly. This is useful if you wish to take advantage of the score player's voice management for something other than playing back a MIDI score (e.g. a sound effects system, see *tsc_soundfx.c* for an example of how to do this). You should note that the functions require a full MIDI environment to be set up, a score context with all of its attendant data structures.

Interpreting and Executing a MIDI Message

To execute any of the MIDI messages recognized by the Music library, use this call:

```
int32 InterpretMIDIMessage( ScoreContext *ScoreCon,
                          char *MIDIMsg, int32 IfMute )
```

This is the call made by *InterpretMIDIEvent()*, which is used as an interpreter procedure for MIDI objects. *InterpretMIDIMessage()* accepts three arguments: **ScoreCon*, a pointer to the score context used to play a MIDI score; **MIDIMsg*, a pointer to a character string containing the MIDI message to be executed; and *IfMute*, a boolean flag that turns muting on and off.

When `InterpretMIDIMessage()` executes, it reads the specified MIDI message and if it recognizes the message, it passes the message and its accompanying data on to the appropriate MIDI function, which executes the message. If the `IfMute` flag is set to `TRUE`, `InterpretMIDIMessage()` does not recognize any Note On messages with velocity values greater than 0. In other words, it does not start any new notes, although it turns notes off and executes other MIDI messages such as program change messages and control change messages.

If successful, `InterpretMIDIMessage()` returns 0. If unsuccessful, it returns a negative value (an error code).

MIDI messages are defined by the MIDI standard as being a series of bytes. The first byte is the message, subsequent bytes (if necessary) are data pertaining to the message. The string that `InterpretMIDIMessage()` accepts is a string of MIDI message bytes, with the MIDI message in the first byte and data bytes following it.

Starting and Releasing a Note

When `InterpretMIDIMessage()` receives a Note On message, it makes this call:

```
int32 StartScoreNote( ScoreContext *scon, int32 Channel,
                     int32 Note, int32 Velocity )
```

The call accepts four arguments: `*scon`, which is a pointer to the score context keeping track of note allocation; `Channel`, which is the channel number (1 to 16) of the note being started; `Note`, which is a MIDI pitch value from 0 to 127; and `Velocity`, which is a MIDI velocity value from 0 to 127.

When it executes, `StartScoreNote()` uses the voice allocation algorithm to find an appropriate instrument and then start a note on that instrument using the specified pitch and velocity. The channel number specified determines which instrument template should be used for the instrument. If successful, it returns 0. If unsuccessful, it returns a negative value (an error code).

When `InterpretMIDIMessage()` receives a Note Off message, it makes this call:

```
int32 ReleaseScoreNote( ScoreContext *scon, int32 Channel,
                       int32 Note, int32 Velocity )
```

The call accepts four arguments: the first, `*scon`, is a pointer to the score context keeping track of note allocation; the second, `Channel`, is the channel number (1 to 16) of the note being released; the third, `Note`, is a MIDI pitch value from 0 to 127; and last, `Velocity`, is a MIDI velocity value from 0 to 127.

When it executes, `ReleaseScoreNote()` finds the note of the specified pitch using the specified channel, and releases the note using the specified velocity. A released note does not necessarily stop playing immediately, so the instrument used for the note may continue to play for a while longer until the note stops and the instrument is abandoned. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

To immediately stop a note, use this call"

```
Err StopScoreNote( ScoreContext *scon, int32 Channel,
                  int32 Note )
```

This call accepts three arguments: the first, `*scon`, is a pointer to a `ScoreContext` data structure controlling playback; the second, `Channel`, is the number of the MIDI channel in which to play the note; and the third, `Note`, is the MIDI pitch value of the note (0-127).

When it executes, `StopScoreNote` immediately stops a note for the score context. It differs from `ReleaseScoreNote()` in that it doesn't allow the note to go through any release phase that it might have. This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Directly Starting and Releasing an Instrument

`ReleaseScoreNote()` and `StartScoreNote()` use voice allocation as they start and release notes. Once a voice is allocated, they call lower level functions to start or release an instrument for the voice.

This call starts a specified instrument without using the MIDI environment's voice allocation mechanism:

```
int32 NoteOnIns( Item Instrument, int32 Note,
                int32 Velocity )
```

The call accepts three arguments: `Instrument`, which is the item number of the instrument to start; `Note`, which is a MIDI pitch value from 0 to 127; and `Velocity`, which is a MIDI velocity value from 0 to 127.

When it executes, `NoteOnIns()` uses the Audio folio to start the specified instrument using the pitch specified by `Note` and the amplitude specified by `Velocity`. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

This call releases a note on a specified instrument without using the MIDI environment's voice allocation mechanism:

```
int32 NoteOffIns( Item Instrument, int32 Note,
                 int32 Velocity )
```


The call accepts three arguments: *Instrument*, which is the item number of the instrument to release; *Note*, which is a MIDI pitch value from 0 to 127; and *Velocity*, which is a MIDI velocity value from 0 to 127.

When executed, `NoteOnIns()` uses the Audio folio to release the specified instrument using the release value specified by *Velocity*. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

Changing a Program

When `InterpretMIDIMessage()` receives a Program Change message, it makes this call:

```
int32 ChangeScoreProgram( ScoreContext *ScoreCon,
                          int32 Channel, int32 ProgramNum )
```

The call accepts three arguments: **ScoreCon*, which is a pointer to the score context controlling the MIDI environment; *Channel*, which is the number of the channel (0 to 15) for which the program should be changed; and *ProgramNum*, a number from 1 to 128 that specifies the new program to be used for the specified channel.

When it executes, `ChangeScoreProgram()` changes the program number associated with a given channel in the MIDI environment. Any future notes played in that channel use the instrument template specified in the PIMap for the new program. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

Setting Channel Panning and Volume

MIDI Control messages set panning and volume levels for each channel in the MIDI environment. When `InterpretMIDIMessage()` receives a Control message, it makes this call:

```
int32 ChangeScoreControl( ScoreContext *scon, int32 Channel,
                          int32 Index, int32 Value )
```

The call accepts four arguments: **scon*, which is a pointer to the score context controlling the MIDI environment; *Channel*, which is the number of the channel (0 to 15) for which a controller should be set; *Index*, which is the number of the controller to be set; and *Value*, which contains the value of the new controller setting.

`ChangeScoreControl()` accepts only two different *Index* values: 7, which specifies a change in volume; and 10, which specifies a change in panning. When `ChangeScoreControl()` executes, it checks the *Index* value. If it is 7, it looks for a value from 0 to 127 in *Value*, which it uses to set the overall volume for the channel. 0 is no volume, 127 is maximum volume.

If the `Index` value is 10, the function looks for a value from 0 to 127 in `Value`, which it uses to set panning for the channel. Panning is the position of the channel's notes in the stereo output signal, which the function sets in the MIDI mixer. 0 is all the way to the left of the output, 63 is the middle of the output, and 127 is all the way to the right of the output.

Bending Pitch

Most dedicated synthesizers have a pitch bend wheel next to the keyboard. When you rotate the pitch bend wheel up, it bends up the pitch of all voices in the current channel. When you rotate the wheel down, it bends down the pitch of all the voices. The amount of pitch bend depends on the pitch bend range set for the synthesizer. The range is often one semitone (a half step), which means that the wheel can bend pitch up by a maximum of one semitone or down by a maximum of one semitone. If the range is set higher, say to an octave, then the pitch wheel can bend pitch much further, an octave in each direction.

When a pitch wheel is turned, it sends a stream of Pitch Bend Change messages with accompanying data values that range from 0x0 to 0x3FFF. These data values describe the positions through which the wheel moves. Value 0x0 means that the wheel is turned all the way down, signifying maximum pitch bend down. Value 0x3FFF means that the wheel is turned all the way up, signifying maximum pitch bend up. And value 0x2000 means that the wheel is set to its center position, so it does not bend pitch at all. When the wheel has stopped moving, it stops sending Pitch Bend Change messages, and pitch bend freezes at the wheel's last reported position.

The actual amount of pitch bend applied to a channel's voices is the result of the MIDI pitch bend value (the wheel's position) applied to the pitch bend range. For example, if the pitch bend value is 0x1000 (half way down to the lowest possible pitch bend value of 0x0) and the pitch bend range is a whole step, then the final pitch bend applied to the channel's voices is down by one half step, one half of the whole step range.

Setting a Pitch Bend Range Value

In the Music library's internal MIDI environment, the current pitch bend range value, which applies to all channels within a score, is stored in the `ScoreContext` data structure. The pitch bend range value is used together with a MIDI pitch bend value to determine how far the pitch of the channel's voices should be bent.

To set a score's pitch bend range value (the maximum distance from normal pitch that any score voices can be bent), use this call:

```
Err SetScoreBendRange( ScoreContext *scon, int32 BendRange )
```

The call accepts two arguments: **scon*, a pointer to the *ScoreContext* data structure in which to set the pitch bend range; and *BendRange*, a pitch bend range value from 1 to 12 that specifies the number of semitones (half steps) in the pitch bend range. A minimum *BendRange* value of 1 means that all voices in a score can be bent up or down a maximum of one half step. And a maximum *BendRange* value of 12 means that all voices in a score can be bent up or down a maximum of one octave (12 half steps).

When it executes, *SetScoreBendRange()* writes the *BendRange* value into the appropriate element of the specified score context. It returns 0 if successful; it returns a negative value (an error code) if unsuccessful.

To see what a score context's current pitch bend range value is, use this call:

```
int32 GetScoreBendRange( ScoreContext *scon )
```

It accepts one argument: **scon*, a pointer to the *ScoreContext* data structure you want to check. When it executes, it returns the current pitch bend range value of the score context if successful. If unsuccessful, it returns a negative value (an error code).

Acting on Pitch Bend Change Messages

Dynamic pitch bending occurs whenever a MIDI score object containing a stream of Pitch Bend Change messages is played back by the Juggler. Those messages are passed on to *InterpretMIDIMessage()*, which makes this call once for each Pitch Bend Change message:

```
Err ChangeScorePitchBend( ScoreContext *scon, int32 Channel,
                          int32 Bend )
```

The call accepts three arguments: **scon*, a pointer to the *ScoreContext* data structure controlling playback; *Channel*, the number of the channel (0 to 15) specified for the pitch bending; and *Bend*, a MIDI pitch bend value from 0x0 to 0x7FFF, which specifies the amount of pitch bend.

ChangeScorePitchBend(), when it executes, reads the score's current pitch bend range value from the specified score context. It then calls *ConvertPitchBend()* and passes it the pitch bend range value and the MIDI pitch bend value (*Bend*). *ConvertPitchBend()* returns an internal pitch bend value, which *ChangeScorePitchBend()* writes to the *ScoreChannel* data structure as the channel's current pitch bend value. *ChangeScorePitchBend()* also calls the audio function *BendInstrumentPitch()* to bend each of the instruments used for the specified channel's voices. Any new instruments created to support the channel's voices are set to bend pitch by the amount specified in the *ScoreChannel* data structure.

If the preceding description seems complicated, consider the final results of `ChangeScorePitchBend()`'s execution: all channel voices are bent to the amount specified in the MIDI message, and the current bend setting is saved until a new bend setting is specified.

If successful, `ChangeScorePitchBend()` returns 0. If unsuccessful, it returns a negative value (an error code).

Note that although `ChangeScorePitchBend()` is usually called by `InterpretMIDIMessage()`, a task can call it directly if it wants to reset a channel's pitch bend.

Creating an Internal Pitch Bend Value

A task can call `ConvertPitchBend()` directly if it wants to translate a MIDI pitch bend value (contained as data in a Pitch Bend Change message) into an internal pitch bend value that can be used by the Audio folio:

```
Err ConvertPitchBend( int32 Bend, int32 SemitoneRange, float32
    *BendFractionPtr )
```

The call accepts three arguments: `Bend`, which is a MIDI pitch bend value from 0x0 to 0x3FFF; `SemitoneRange`, which is the pitch bend range value measured in semitones (half steps); and `*BendFractionPtr`, which is a pointer to a `frac16` variable in which to store the internal pitch bend value calculated by the function.

When it executes, `ConvertPitchBend()` applies the MIDI pitch bend value to the pitch bend range to see just how far pitch must be bent. It calculates an internal pitch bend value appropriate for that much pitch bend, and writes it into the `frac16` variable. If successful, it returns 0; if unsuccessful, it returns a negative value (an error code). The `BendFraction` can then be used with `BendInstrumentPitch()`.

Changing Characteristics During Playback

A task can play a MIDI score using the Juggler, or it can play notes by directly calling MIDI playback functions. A third option is for the task to play a MIDI score using the Juggler and make direct calls to MIDI playback functions during the score playback to change playback characteristics. For example, a task can call `ChangeScoreControl()` during playback to set panning for one channel, effectively moving the notes played in that channel from one side of the stereo output to the other.

All of the MIDI playback functions can safely be called directly during score playback.

Cleaning Up

When a task finishes playing a MIDI score and does not need the MIDI environment anymore, it should clean up behind itself. It should free allocated amplitude, disown the audio clock and reset it to its original rate, get rid of the score context and associated data structures, delete any instrument templates previously loaded (thus freeing any instruments associated with them), destroy the MIDI objects created to hold the score, terminate the Juggler, and then close the Math and Audio folios.

To free amplitude allocated for the MIDI mixer, use `FreeAmplitude()`. To reset the audio clock use `SetAudioRate()`. And to disown the audio clock, use `DisownAudioClock()`. Both clock calls are described in Chapter 6 “Advanced Audio Folio Usage.”

If your task loaded instrument templates using the `LoadPIMap()` call, you can delete those instrument templates using this call:

```
int32 UnloadPIMap( ScoreContext *scon )
```

It accepts a single argument: `*scon`, a pointer to the score context using the PIMap. When it executes, it deletes all the instrument templates listed in the PIMap. By deleting all the instrument templates, it also deletes any instruments that were created using those templates. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

To delete a Juggler sequence used to store a format 0 MIDI score, use `DestroyObject()`. To delete a Juggler collection used to store a Format 1 MIDI score (along with all sequences contained in the collection), use this call:

```
int32 MFUnloadCollection( Collection *ColPtr )
```

The call accepts one argument: `*ColPtr`, a pointer to the collection to be deleted. When it executes, it destroys the collection and any sequences defined as part of the collection. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

To delete the mixer instrument template (and the mixer instrument) used to mix the voices of a score context, use this call:

```
int32 TermScoreMixer( ScoreContext *ScoreCon )
```

It accepts one argument: `*Scorecon`, a pointer to the score context using the mixer. When it executes, it deletes the instrument template of the mixer specified by the score context, which also deletes the mixer instrument created using the template. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

To delete a score context, use this call:

```
int32 DeleteScoreContext( ScoreContext *scon )
```

It accepts one argument: `*scon`, a pointer to the score context to be deleted. When it executes, it deletes the score context and all of its attendant data structures, including `NoteTrackers`. If successful, the call returns 0. If unsuccessful, it returns a negative value (an error code).

To terminate the Juggler, use the `TermJuggler()` call (see Chapter 10, "Creating and Playing Juggler Objects"). To close the Math and Audio folios, use the `CloseAudioFolio()` and `CloseMathFolio()` calls.

Creating a MIDI Score for Playback

Composing a score usually takes place in an application such as a sequencer, MIDI recorder, or score composition tool. The source does not really matter, as long as it is conducive to composition, and as long as it can export the music created as a Format 0 or Format 1 MIDI file. Think of Format 0 and Format 1 as the ASCII of the music world, readable by the Portfolio Music library as well as by almost any other MIDI environment. If your composition application can only save its scores in its own native format and not in Format 0 or Format 1, then you cannot use its scores with the Music library.

Before you save a score as a Format 0 or Format 1 file, you may want to go through it to optimize it for Music library playback. Reduce the number of programs; if some are not used to good effect, get rid of them. When you choose program numbers, try to use the lowest numbers possible. This allows you to use a smaller `PIMap` and to support fewer programs in the Music library's MIDI environment, which saves resources. Also reduce the number of simultaneous voices if possible, perhaps by rewriting sections that boost the maximum number of voices for just a few seconds.

After you have tweaked your score and saved it, you should create a text `PIMap` that specifies instrument templates and samples that are as close as possible to the programs in the original MIDI score. You may want to make custom samples to correspond to the sounds you want, or use `ARIA` to create custom instruments that sound as much as possible like the instruments originally used for composition. And whenever possible, use simpler instruments in place of complex instruments to conserve DSP resources.

If you create a MIDI score that is looped, watch for sequences that end with a rest. Create a silent MIDI event to fill in for that rest. If you leave it blank (no MIDI message there), playback on looping can be out of sync because the first MIDI event of the score begins where the rest should be.

Once the score and `PIMap` are set, you can turn them over to the Music library with the code you have written, and then wait for the music to play. (Providing, of course, that you have worked out all the bugs...)

See *Examples/Audio/Score/playmf.c* for an example of playing a MIDI score.

Primary Data Structures

The `MIDIEvent` data structure is used for storing MIDI events within a Juggler sequence used for MIDI score playback.

```
typedef struct MIDIEvent
{
    uint32 mev_Time;
    unsigned char mev_Command;
    unsigned char mev_Data1;
    unsigned char mev_Data2;
    unsigned char mev_Data3;
} MIDIEvent;
```

Function Calls

These Music library function calls import and play back MIDI score files.

MIDI Environment Calls

The following calls control setup of the MIDI environment:

<code>CreateScoreContext()</code>	Allocates a score context.
<code>DeleteScoreContext()</code>	Deletes a score context.
<code>InitScoreDynamics()</code>	Sets up dynamic voice allocation.
<code>InitScoreMixer()</code>	Creates and initializes a mixer instrument for MIDI score playback.
<code>LoadPIMap()</code>	Loads a program-instrument map from a text file.
<code>SetPIMapEntry()</code>	Specifies the instrument to use when a MIDI program change occurs.
<code>TermScoreMixer()</code>	Dispose of mixer created by <code>InitScoreMixer()</code> .

<code>UnloadPIMap()</code>	Unloads instrument templates loaded previously with PIMap file.
----------------------------	---

MIDI Score Calls

The following calls are used to handle MIDI scores:

<code>InterpretMIDIEvent()</code>	Interprets MIDI events within a MIDI object.
<code>MFDefineCollection()</code>	Creates a Juggler collection from a MIDI file image in RAM.
<code>MFLoadCollection()</code>	Loads a set of sequences from a MIDI file.
<code>MFLoadSequence()</code>	Loads a sequence from a MIDI file.
<code>MFUnloadCollection()</code>	Unloads a MIDI collection.

MIDI Playback Calls

The following calls handle playback of MIDI:

<code>ChangeScoreControl()</code>	Changes a MIDI control value for a channel.
<code>ChangeScoreProgram()</code>	Changes the MIDI program for a channel.
<code>InterpretMIDIMessage()</code>	Executes a MIDI message.
<code>NoteOffIns()</code>	Turns off a note played by an instrument.
<code>NoteOnIns()</code>	Turns on a note for an instrument.
<code>ReleaseScoreNote()</code>	Releases a MIDI note, uses voice allocation.
<code>StartScoreNote()</code>	Starts a MIDI note, uses voice allocation.
<code>StopScoreNote()</code>	Stop a MIDI note without releasing it. Uses voice allocation.

Creating and Playing Juggler Objects

This chapter describes the object-oriented programming environment of the Juggler, which is a component of the Music library. The chapter describes the default Juggler object classes (sequence and collection), how to create sequences and collections, and the process used to play a sequence or a collection.

This chapter contains the following topics:

Topic	Page Number
Object-Oriented Programming Concepts	206
Managing Juggler Objects	208
Default Juggler Classes	210
Creating Sequences and Collections	212
Sending Messages to Sequences and Collections	220
Playing Sequences and Collections	225
An Example Program	228
Function Calls and Method Macros	232
Object-Defining Tag Arguments	233

The *Juggler* provides event-handling tools for playing and combining audio and nonaudio events. The Audio folio controls the shape and quality of individual audio events, see Chapter 2, “Understanding 3DO Audio,” for more information on the Audio folio.

The Juggler is an object-oriented programming environment that allows a task to assemble lists of timed events and play them back in sequence or in parallel. The Juggler is much smaller than typical object-oriented systems, and has been optimized for real-time performance.

The Juggler is most often used to assemble and play back audio events, but it can also play back nonaudio events such as graphics and cel-drawing events. A task can use the Juggler to mix audio, graphic, system, and other events in an assembly of events that the Juggler plays in coordinated time.

One of the Juggler's main purposes is to play back MIDI scores. The scores must first be translated into Juggler objects, which the Juggler can then play. Chapter 11, "Playing MIDI Scores," describes the Music library tools that perform MIDI-to-Juggler translations. If you are working strictly with MIDI and do not need to create custom events, you may want to skip to "Playing Sequences and Collections" on page 225. However, the information in this chapter may help you understand how the MIDI score translation calls work.

Note: *In the current release of Portfolio, the Juggler is primarily devoted to MIDI score playing.*

Object-Oriented Programming Concepts

This section provides a brief description of object-oriented programming concepts. If you are familiar with object-oriented programming, you may want to skip to "Managing Juggler Objects" on page 208.

Objects

An *object* is a data structure with associated functions. A task can create an object and then work with it as a self-contained operating entity, similar to a machine whose inner workings are concealed. The task sends a message to the object, which interprets the message using one of its functions. The function performs the desired action and can change variables stored within the object to keep track of what it has done.

Methods and Messages

Each function associated with an object is called a *method*. An object can have one or more methods. A task sends a *message* to an object that indicates which method to execute. A message contains arguments that determine how the method performs.

A task must supply the proper arguments with the message in order for the method to execute. When an object receives a message with its accompanying arguments, it takes over operation, executing the method and dealing with the results of execution.

Object Variables

An object can also have variables associated with it that describe the state of the object. When a method executes, the object variables can be used to determine what actions to perform. A method can also modify object variables to reflect changes it has made to the object.

Object variables are internal to the object, and should not be touched directly by a task working with an object.

Classes and Instances

A *class* defines an object. A class specifies an object's variables and methods and defines how those methods affect the object variables. When a task creates an object, it specifies which class to use for the object's definition. The class is used to create an *instance* of the class. An instance is a single implementation of the class (i.e. an object defined by the class).

Each instance of a class has its own data space and variables that record the state of the instance. The instance also contains pointers to the methods defined by the class. When an object receives a message, the object calls one of the class methods which executes using the calling object's variables.

For example, consider a class that defines an employee object. The object has several variables that record the employee's name, date of hire, date of termination, and salary or hourly rate. The object has one method which accepts a *Pay Rate* message. When the method executes, it prints the employee's rate of pay or the termination date if the employee has left the company.

If a program creates two employee objects that have the following variable values:

Table 12-1 *Employee object variable values.*

employee1	employee2
John Smith	Jane Doe
10/21/90	2/23/92
\$20/hr	\$40,000/yr
3/3/94	null

If a *Pay Rate* message is sent to each object, the employee1 object prints the date that John Smith left the company. The employee2 object prints the annual salary for Jane Doe. Identical messages to objects of the same class can produce different results when the state of the object variables is different.

Creating New Classes

A task can create new classes based on existing classes. These new classes define new types of objects. The new class (called a *subclass* of the existing class) inherits the properties of the existing class (called a *superclass* of the new class). The inherited properties include all the object variables and methods available to the superclass.

The subclass definition adds new object variables and methods to the existing variables and methods of the superclass, and creates a new object definition. The subclass can also change the function behind any of the superclass's methods so the method operates differently in instances of the subclass.

For example, a program defines a new class based on the employee object from the previous example. The new class has, in addition to the existing variables, new variables that record the employee's department, home address, home phone number, and number of vacation days accrued. The class also has a new method that accepts a *Vacation* message. When the method executes, it adds one to the number of vacation days stored in the object.

The new class defines an object that stores more detailed information about an employee and accepts two types of messages: *Pay Rate* or *Vacation*.

Creating new classes based on existing classes saves programming work because each new class does not have to be completely defined; the new class inherits the properties of the superclass and only the new elements need to be defined. A task can build a complete hierarchy of classes, starting with a root class at the top of the hierarchy and descending many levels of subclasses. A subclass at any level retains the methods and variables of all its superclasses, all the way back to the root class.

Managing Juggler Objects

A group of Music library functions allows a task to manage Juggler objects. The library functions allow a task to initialize the Juggler, define new classes, create objects from those classes, destroy existing objects, and check an object's validity.

Initializing the Juggler

Before a task can perform any object-oriented work in the Juggler environment, including object management and playing objects with the Juggler, the task must first initialize the Juggler with the following call:

```
int32 InitJuggler( void )
```

This call accepts no arguments and, when it executes, initializes the Juggler and its object-oriented programming environment. It also defines the Juggler default classes so the task can create objects or define new classes based on the default classes.

If successful, `InitJuggler()` returns 0. If unsuccessful, it returns an error code (a negative value).

Note that a task *must* call `InitJuggler()` before it can use any Juggler calls or play Juggler objects (including MIDI scores created as described in Chapter 11, "Playing MIDI Scores").

Defining a New Class

Once a task has initialized the Juggler, the task can define a new class based on the Juggler default classes (described in "Default Juggler Classes" on page 210). To define a new class, the task uses this call:

```
int32 DefineClass( COBClass *Class, COBClass *SuperClass,
                  int32 DataSize )
```

Creating an Object

To create an object based on an existing class, a task uses this call:

```
COBObject *CreateObject( COBClass *Class )
```

The call accepts a single argument: `*Class`, a pointer to the `COBClass` data structure containing the class definition. There are currently two classes defined in the header file `juggler.h`. The classes are `SequenceClass` and `CollectionClass`, which you can supply to this call as the pointers `&SequenceClass` and `&CollectionClass`.

When it executes, `CreateObject()` allocates memory for the object, sets up object variables and method pointers in that memory, and initializes the object variables to default values.

`CreateObject()` returns a pointer to the `COBObject` data structure defining the object if successful or a negative value (an error code) if unsuccessful. The task uses the pointer to the `COBObject` structure when sending messages to the object or when using other object management calls on the object.

Destroying an Object

When a task is finished with an object, it deletes the object from memory using this call:

```
int32 DestroyObject( COBObject *Object )
```

The call accepts one argument: `*Object`, a pointer to the object to be destroyed. When executed, the call terminates any methods the object may be executing and frees the memory used to contain the object's `COBObject` data structure.

`DestroyObject()` returns 0 if successful, or a negative value (an error code) if unsuccessful.

Checking an Object's Validity

A task must pass a pointer to an object to an object-management call or a method. If the argument does not point to an object, the results of the call are unpredictable. A task can check a pointer's validity using this call:

```
int32 ValidateObject( COBObject *cob )
```

The call accepts one argument: `*cob`, a pointer to an object's `COBObject` data structure. When the call executes, it looks in the object's data structure for an element that confirms the object's validity.

If the object is valid, `ValidateObject()` returns 0. Otherwise, the call returns a negative value (an error code).

Default Juggler Classes

The Juggler currently provides three default object classes:

- ◆ The *jugglee class*, the root class for the entire class hierarchy.
- ◆ The *sequence class*, a jugglee subclass that creates an object that can maintain and play back a sequence of events.
- ◆ The *collection class*, a jugglee subclass that creates an object that can maintain and play back in parallel an assembly of sequences and other collections.

These three classes are defined in the header file *juggler.h*.

The Jugglee Class

The jugglee class is the root class for all other Juggler classes. Its elements include pointers to a set of methods common to all Juggler objects. Those methods are discussed in "Message Macros" on page 220.

The jugglee class also includes variables that store information about the object, including the object size, a validation key, hierarchical ties, the start time for the object, the time of the next event to play, a count stating how many times the object should be repeated, a flag showing whether the object is active or not, and delay values used to set delays before an object starts playing, repeats, or stops.

Note: *The Juggler does not currently support the creation of custom classes. The class variables are handled by the class instance.*

The jugglee class is designed as a prototype for other Juggler classes; it should only be used to create new classes. *Do not* use the jugglee class to create jugglee objects. A jugglee object does not have the necessary elements to play back events.

The Sequence Class

The sequence class is used to assemble and play sequential events. The events can be audio or nonaudio. The sequence class contains all of the jugglee's methods and variables and also has a pointer to an *event list*, a list of timed events and a pointer to an *interpreter procedure*, which interprets each event in the event list.

Event Lists

Each event in the list is a time value followed by one or more data values. An example of an event list is shown in Table 12-2.

Table 12-2 *Example event list.*

Time	Data
0	56
120	58
480	54

The time for each event is measured in ticks, an arbitrary unit of time. The time is a relative value, the amount of time in ticks that elapses from the beginning of playback until the event starts. For example, the first event in Table 12-2 starts as soon as the sequence begins playback. The second event starts 120 ticks after the start of the sequence, and the third event starts 480 ticks after the start of the sequence.

The task playing the sequence supplies timing by giving tick values directly to the Juggler, which passes it on to the sequence for playback. See "Bumping the Juggler" on page 227 for information on supplying tick values to the Juggler. The audio timer is commonly used for tick values, which ticks approximately 240 times per second at default speed. If the event list in Table 12-2 is played using audio timer values, the second event starts one-half second after the first event (120 ticks), and the third event starts one-and-one-half seconds after the second event ($480 - 120 = 360$ ticks).

The Interpreter Procedure

The event data associated with each event can be a single value, as shown in Table 12-2, or it can be multiple values. This data has meaning only to the interpreter procedure associated with the sequence. When the sequence is played, the data is passed to the interpreter procedure, which executes according to that data.

The task that creates a sequence object creates the interpreter procedure and the data to go with it; a task can create a sequence containing almost any kind of event.

For example, an interpreter procedure can play half-second-long notes on an audio instrument. It accepts event data and interprets the data as a MIDI pitch value, which sets the pitch of each instrument note played. The interpreter procedure can read pitch, amplitude, and duration for each event and play audio instrument notes accordingly. It can accept other types of data, such as two sets of coordinates that it uses to project a cel onscreen.

The Collection Class

The collection class is used to create higher-level event structures: assemblies of sequences, assemblies of other collections, or assemblies of sequences and collections combined.

The collection class contains all of the juggler's methods and variables, and also contains a list of *placeholders*. Each placeholder contains a pointer to a Juggler object and a variable telling how many times to repeat the object in playback. The list of placeholders, called the *object list*, defines the collection's assembly of objects.

The collection class also contains new variables in addition to the juggler class variables. The new variables store the current time (used to determine what events need to be played) and the time of the next event to be played (used to report to the juggler).

When a collection is played, its objects are played in parallel and each object is repeated the number of times specified by the object's placeholder. Parallel playback means that each object starts playback simultaneously when the collection starts playback.

Creating Sequences and Collections

To create a sequence or collection, a task must do the following:

- ◆ Create auxiliary components if they do not exist: an event list and interpreter procedure for a sequence, subsidiary sequences and collections for a collection.

- ◆ Create an instance of a sequence or a collection.
- ◆ Create a list of tag arguments to set object variable values.
- ◆ Apply the tag argument list to the sequence or collection. This step is required for sequences; pointers to the event list and interpreter procedure must be set. This step is not always required for collections.
- ◆ Add objects to a collection's object list.

This remainder of this section describes the steps for creating a sequence and a collection. It also lists the tag arguments used to set variables for each object type.

Creating a Sequence

Before creating a sequence, a task should create an event list that contains the event data. The task should also create an interpreter procedure appropriate for event list.

Creating an Event List

An event list is an array of event data structures. The event data structure contains a time value as its first element, followed by as many data elements as the interpreter procedure requires. For example, the data structure shown in Example 12-1 defines a simple event consisting of a time and a single data value.

Example 12-1 *Simple event structure.*

```
typedef struct
{
    Time      te_TimeStamp;
    uint32    te_Data;
} SimpleEvent;
```

Example 12-2 shows an array of simple event structures:

Example 12-2 *Array of simple event structures.*

```
SimpleEvent  TimeData[] =
{
    { 0, 56 },
    { 120, 58 },
    { 480, 54 }
};
```

Example 12-3 shows a more complex event structure that could be used for graphics events.

Example 12-3 *Example of a custom graphics event.*

```
typedef struct CustomGraphicsEvent
{
    uint32    cge_Time;
    float32    cge_XPos;    /* Where to draw M2 Frame Object */
    float32    cge_YPos;
    float32    cge_ZPos;
    floatCCB    *cge_M2 Frame Object;
} CustomGraphicsEvent;
```

Creating the Interpreter Procedure

When the Juggler plays a sequence, the sequence sends each event to be played to the interpreter procedure. The interpreter procedure receives two pointers for each event: a pointer to the sequence object sending the event and a pointer to the event data structure. The interpreter procedure can examine the sequence object's variables or the data in the event data structure before playing the event.

Example 12-4 shows a simple interpreter procedure. This procedure prints the sequence's address, the timestamp stored in the event, and the single data value stored in the event.

Example 12-4 *Simple interpreter procedure.*

```
int32 UserInterpFunc ( Jugglee *Self, SimpleEvent *se )
{
    PRT(("TestEvent: Self = 0x%x, Time = %d, Data = %d\n", Self,
        se->se_TimeStamp, se->se_Data));
    return 0;
}
```

Not all interpreter procedures play audio events. The interpreter procedure in Example 12-4 plays print events.

Creating an Instance of a Sequence

create an instance of a sequence, the task uses the `CreateObject()` function. In Example 12-5, the task supplies a pointer to the sequence class, and stores the return value, a pointer to the newly created sequence, in the variable `seq1`.

Example 12-5 *Creating an instance of a sequence.*

```
seq1 = (Sequence *) CreateObject( &SequenceClass );
```

Note that `SequenceClass` is defined in the *juggler.h* header file. You must include *juggler.h* in code that creates a sequence.

Setting Sequence Variables With a Tag Argument List

When a sequence is created with `CreateObject()`, its object variables are set to default values. A task can change those values by creating a tag argument list and applying the list to the object.

A task *must* set the sequence variables that point to or describe the event list or the sequence cannot be played.

A task must set the following tag argument values for a sequence:

- ◆ `JGLR_TAG_EVENTS` is a pointer to the array containing the event list. The default setting for this tag is `NULL`.
- ◆ `JGLR_TAG_EVENT_SIZE` is an integer specifying the size in bytes of a single event in the event list. The default setting for this tag is `NULL`.
- ◆ `JGLR_TAG_MAX` is an integer specifying the maximum number of events that can be stored in the event list. This number is usually determined by the amount of memory allocated for the event list. The default setting for this tag is `NULL`.
- ◆ `JGLR_TAG_MANY` is an integer specifying the current number of events stored in the event list. The current number of events can be less than the maximum number of events. The task should change this value if the task changes the number of events in the list. The default setting for this tag is `NULL`.
- ◆ `JGLR_TAG_INTERPRETER_FUNCTION` is a pointer to the interpreter procedure. The default setting for this tag is `NULL`.

Sequence variables can also be set using the following tag arguments:

- ◆ `JGLR_TAG_START_FUNCTION` is a pointer to a function that executes when the sequence is started. This function is useful for setting up conditions for the sequence—for example, a function that clears the screen for a cel-drawing sequence. The default setting for this tag is `NULL`.
- ◆ `JGLR_TAG_REPEAT_FUNCTION` is a pointer to a function that executes immediately before a sequence is repeated. This function is useful for resetting conditions for the sequence. The default setting for this tag is `NULL`.

- ◆ JGLR_TAG_STOP_FUNCTION is a pointer to a function that executes immediately after a sequence stops. This function is useful for restoring conditions after a sequence is played. The default setting for this tag is NULL.
- ◆ JGLR_TAG_START_DELAY is an integer that stores the number of ticks the sequence should wait after playback begins before it starts playing its events. The default setting for this tag is NULL.
- ◆ JGLR_TAG_REPEAT_DELAY is an integer that stores the number of ticks the sequence should wait before repeating playback (useful if the sequence is to be repeated several times). The default setting for this tag is NULL.
- ◆ JGLR_TAG_STOP_DELAY is an integer that stores the number of ticks the sequence should wait, after it stops playback (useful if the sequence is one of several stored one after another within a collection). The default setting for this tag is NULL.
- ◆ JGLR_TAG_SELECTOR_FUNCTION The default setting for this tag is NULL.
- ◆ JGLR_TAG_MUTE is a flag used for MIDI score playback. If the flag is TRUE, then no MIDI Note On messages in the sequence are played. If the flag is FALSE, then MIDI Note On messages are played. The default setting for this tag is FALSE.
- ◆ JGLR_TAG_CONTEXT is a pointer to an optional user context. The user context is a data structure that contains information about the context within which the sequence plays. The pointer to the user context can be used by the interpreter procedure. A user context is employed by the MIDI interpretation tools described in Chapter 11, "Playing MIDI Scores," but is not typically used by tasks creating sequences. The default setting for this tag is NULL.

The task that created a sequence creates a tag argument list for the sequence using an array of the TagArg elements (the same data type used for item tag arguments). Example 12-6 shows an example of code to create a tag argument list.

Example 12-6 *Creating a tag argument list.*

```
TagArg tags[NTAGS];

tags[0].ta_Tag = JGLR_TAG_CONTEXT;
tags[0].ta_Arg = (TagData)scorecontext;
tags[1].ta_Tag = JGLR_TAG_MUTE;
tags[1].ta_Arg = (TagData)FALSE;
tags[2].ta_Tag = TAG_END;
```

Applying Tag Argument Values to a Sequence

After creating a tag argument list, the task applies them to the sequence by calling the sequence's `SetInfo` method. The following macro calls the `SetInfo` method:

```
int32 SetObjectInfo( CObject *obj, TagArg *tags )
```

See "SetInfo" on page 221 for details on the `SetInfo` method.

Reading Tag Argument Values

To find out what the object variable values for an object are, a task creates a tag argument list with each tag argument value set to `NULL`. It then uses the following macro, which calls an object's `GetInfo` method to obtain the variable values:

```
int32 GetObjectInfo( CObject *obj, TagArg *tags )
```

See "GetInfo" on page 221 for details on the `GetInfo` method.

Creating a Collection

Creating a collection is similar to creating a sequence, but a collection creates an assembly of Juggler objects in its object list instead of a sequence of events in an event list. The object list is part of the collection object. A sequence's event list, which is a data structure outside a sequence object, can be manipulated directly by a task. A collection's object list is created using collection method calls.

Creating Objects for the Collection

A collection must have Juggler objects to include in its object list. Therefore, a task must first create some sequences before it can create a collection. After a task has created one or more collections, it can create new collections that include existing collections.

Creating an Instance of a Collection

To create an instance of the collection class, the task uses the `CreateObject()` function. The task supplies a pointer to the collection class, and stores the pointer to the newly created collection as shown in this example:

```
coll = (Collection *) CreateObject( &CollectionClass );
```

Note that `CollectionClass` is defined in the *juggler.h* header file. You must include *juggler.h* in code that creates a sequence.

Setting Collection Variables With Tag Argument Values

A collection's object variables can be changed using a list of tag argument values. However, most collections can use the default values. Collections accept the following tag arguments:

- ◆ **JGLR_START_FUNCTION** is a pointer to a function that executes when the collection is started. The function is useful for setting conditions before the collection starts playback. The default setting for this tag is NULL.
- ◆ **JGLR_TAG_REPEAT_FUNCTION** is a pointer to a function that executes immediately before a collection is repeated. This function is useful for resetting conditions for the collection. It can also be used to chain music together by calling `StartObject()` for another object. See the example file *MFLoopTest.c*. The default setting for this tag is NULL.
- ◆ **JGLR_TAG_STOP_FUNCTION** is a pointer to a function that executes immediately after a collection stops. This function is useful for restoring conditions after collection playback. The default setting for this tag is NULL.
- ◆ **JGLR_TAG_START_DELAY** is an integer that stores the number of ticks the collection should wait before it starts playing back its events. The default setting for this tag is NULL.
- ◆ **JGLR_TAG_REPEAT_DELAY** is an integer that stores the number of ticks the collection should wait before repeating playback (useful if the collection is asked to repeat playback several times). The default setting for this tag is NULL.
- ◆ **JGLR_TAG_STOP_DELAY** is an integer that stores the number of ticks the collection should wait after it stops playback (useful if the collection is one of several collections stored one after another within another collection). The default setting for this tag is NULL.
- ◆ **JGLR_TAG_CONTEXT** is a pointer to an optional user context. The user context is a data structure that contains information about the context within which the collection plays. It can be used by the interpreter procedure. The default setting for this tag is NULL.

After a task has created tag argument values for a collection, it applies them with the `SetObjectInfo()` macro. To obtain current object variable values from a collection, a task uses the `GetObjectInfo()` macro.

Adding Objects to the Collection

After a task creates a collection and (if necessary) sets object variables with a tag argument list, it can then add objects to the collection using the collection's `Add` method. No macro currently exists for the `Add` method; it must be called directly by using its pointer within the collection, as follows:

```
*obj->Class->Add( Collection *Self, Jugglee *Child, int32
                  NumRepeats)
```

To call the method, `*obj` must be a pointer to the collection. The call accepts three arguments: `*Self`, a pointer to the collection accepting the object; `*Child`, a pointer to the object to be added to the collection; and `NumRepeats`, an integer specifying the number of times the added object should be repeated in playback.

When the `Add` method is executed, it adds the specified object to the end of the collection's object list and writes the number of repeats into the object's placeholder. The `Add` method returns 0 if successful, or a negative value (an error code) if unsuccessful.

The following example adds a sequence with the pointer `sequencex` to the object list of a collection specified by the pointer `collectionx`. The sequence is repeated three times during collection playback.

```
Result = collectionx->Class->Add( collectionx, sequencex, 3 );
```

Removing Objects From a Collection

To remove an object from a collection, a task uses the collection's `RemoveNthFrom` method, which is called using this macro:

```
int32 RemoveNthFromObject( CObject *obj, int32 N )
```

The call accepts two arguments: `*obj`, a pointer to the collection from which the object should be removed; and `N`, an integer that specifies the placeholder in the object list that points to the object to be removed. The first object in the placeholder list is object number 0. For example, the placeholder for the fifth object in an object list is 4.

When `RemoveNthFromObject()` executes, it removes the specified placeholder from the object list. It does *not* do anything to the object itself.

If successful, `RemoveNthFromObject()` returns 0. If unsuccessful, it returns a negative value (an error code).

Examining a Collection's Object List

To see what objects are included in a collection's object list, a task uses the collection's `GetNthFrom` method, which is called by this macro:

```
int32 GetNthFromObject( CObject *obj, int32 N, (void**) NthThing
                        )
```

The call accepts three arguments: `*obj`, a pointer to the collection whose list is to be checked; `N`, the number of the placeholder to read in the object list; and `NthThing`, a pointer to a variable in which to store a pointer.

When it executes, the call looks at the Nth placeholder in the specified collection's object list. It finds the pointer stored in the placeholder and writes the pointer to the NthThing variable.

GetNthFromObject () returns 0 if successful. If unsuccessful, it returns a negative value (an error code).

The first object in the placeholder list is object number 0, the second is 1, and so on.

Sending Messages to Sequences and Collections

After a task creates a Juggler object, it can send messages to the object. This section describes the messages that sequences and collections accept and also describes the actions of the methods in response to the messages.

The function for a method can be different for different types of objects. Therefore, the results of identical messages sent to different objects can vary.

Calling Methods Directly

Methods are defined within an object with a list of pointers to functions. A task can call any of these methods directly using the pointer to the method's function that is in the object data structure. For example, *obj->Class->Add calls an object's Add method.

Message Macros

Many of the methods available to Juggler objects can be called through macros provided by the Music library. To make your code more legible, you should use macros whenever possible. For example, the macro StopObject () calls the method *obj->Class->Stop. See "Method Macros" on page 233 for a list of macros.

Messages for Sequences

Although a sequence accepts messages for all the common Juggler methods, not all methods have meaningful results for a sequence. For example, a sequence accepts an Add message to add an object to an object list, but the message does not do anything meaningful because the sequence has no object list. The following sections describe the methods that give meaningful results for a sequence.

Alloc

The Alloc method allocates memory for a sequence's event list. The Alloc method is called using this macro:

```
int32 AllocObject( CObject *obj, int32 Num )
```


The call accepts two arguments: **obj*, a pointer to the sequence for which memory should be allocated; and *Num*, an integer that specifies the number of events in the event list.

When it executes, the call uses the event size stored in the sequence to allocate an appropriate amount of RAM for the event list. It stores a pointer to the allocated RAM into the object variable **obj->seq_Events*. `AllocObject()` returns 0 if successful or a negative value (an error code) if unsuccessful.

A task does not have to use this method to allocate memory for an event list. The method is provided as a convenience. A task can use its own allocation techniques.

Free

The Free method frees memory previously allocated for a sequence's event list by the sequence's Alloc method. The Free method is called by this macro:

```
int32 FreeObject( CObject *obj )
```

The call accepts one argument: **obj*, a pointer to a sequence.

When it executes, the call frees any memory previously allocated for the sequence's event list. If successful, `FreeObject()` returns 0. If unsuccessful, it returns a negative value (an error code).

If a task allocates memory for an event list using its own techniques, the memory should be freed using its own techniques.

SetInfo

The SetInfo method sets a sequence's variables with values provided by a tag argument list. The SetInfo method is called by the following macro:

```
int32 SetObjectInfo( CObject *obj, TagArg *tags )
```

The call accepts two arguments: **obj*, a pointer to the object to which the tag argument values should be applied; and **tags*, a pointer to the tag argument list. When executed, the call sends a SetInfo message to the specified object. The object reads the values in the tag argument list and applies them to the appropriate object variables.

`SetObjectInfo()` returns 0 if successful, or a negative value (an error code) if unsuccessful.

GetInfo

The GetInfo method returns a sequence's variable values by writing them to a provided tag argument list. It's called by this macro:

```
int32 GetObjectInfo( CObject *obj, TagArg *tags )
```

The call accepts two arguments: **obj*, a pointer to the object for which values should be retrieved; and **tags*, a pointer to a tag argument list. The call sends a *GetInfo* message to the specified object. The object writes its current variable values into the tag argument list.

GetObjectInfo() returns 0 if successful, and a negative value (an error code) if unsuccessful.

Start

The *Start* method makes a sequence active so that it can be played by the Juggler when the Juggler is bumped (See "Bumping the Juggler" on page 227). The method is called by the following macro:

```
int32 StartObject( CObject *obj, uint32 Time, int32 NumReps,
                  CObject *Parent )
```

The call accepts four arguments: **obj*, a pointer to the sequence to be started; *Time*, an absolute time in ticks when the sequence should be started; *NumReps*, the number of times the sequence should be repeated in playback; and **Parent*, a pointer to the parent of the sequence.

The **Parent* pointer is only used when a sequence is part of a collection. A task using this call directly should set **Parent* to *NULL*. The *NumReps* value only sets the number of playback repetitions for a sequence if the sequence is not part of a collection. If it is part of a collection, the number of repetitions stored in the sequence's placeholder overrides this value.

When *StartObject()* executes, the call stores the *Time* and *NumReps* values in the appropriate sequence variables. The Juggler uses the start time to determine the absolute event times (See "Absolute and Relative Event Times" on page 225). The Juggler uses the *NumReps* value to determine how many times to play the sequence.

StartObject() returns 0 if successful, and a negative value (an error code) if unsuccessful.

Stop

The *Stop* method makes a sequence inactive (it is no longer to be played by the Juggler). The *Stop* method is called by this macro:

```
int32 StopObject( CObject *obj, uint32 Time )
```

The call accepts two arguments: **obj*, a pointer to the sequence to be stopped; and *Time*, an absolute time in ticks that reports when the object was stopped.

When executed, the call makes the sequence inactive so that none of its events are played by the Juggler. The reported stop time is passed by the stopped object to any existing parent objects. `StopObject()` returns 0 if successful, and a negative value (an error code) if unsuccessful.

Note: *A task should be careful when stopping a sequence contained within a collection. The sequence may stop for a while, but if it is repeated by the collection or called by another placeholder, it may start again. In other words, stopping a sequence within a collection may have unpredictable results.*

Print

The Print method prints debugging information about a sequence. The Print method is called by this macro:

```
int32 PrintObject( CObject *obj )
```

The call accepts one argument: `*obj`, a pointer to the sequence for which information should be printed.

When executed, the call prints debugging information about the specified sequence, including a pointer to the sequence. `PrintObject()` returns 0 if successful, and a negative value (an error code) if unsuccessful.

Messages for Collections

A collection accepts all the common jugglee class methods, but only some of them have meaningful results for a collection. The following sections describe the methods that have meaningful results for a collection.

Alloc and Free

These methods have no meaningful results for a collection.

SetInfo and GetInfo

These methods, called by the macros `SetObjectInfo()` and `GetObjectInfo()`, perform the same actions as they do for a sequence. See "SetInfo" on page 221 and "GetInfo" on page 221.

Add

The Add method adds a Juggler object to a collection's object list. There is no macro for this method. The Add method is called directly, as follows:

```
*obj->Class->Add( Collection *Self, Jugglee *Child,  
int32 NumRepeats)
```

See "Adding Objects to the Collection" on page 218 for more information on the Add method.

GetNthFrom

The GetNthFrom method looks at a specified position in a collection's object list and retrieves a pointer to the Juggler object. The GetNthFrom method is called by the following macro:

```
int32 GetNthFromObject( CObject *obj, int32 N, (void**) NthThing
)
```

See "Examining a Collection's Object List" on page 219 for more information on the GetNthFrom method.

RemoveNthFrom

The RemoveNthFrom method removes the object from a specified location within a collection object list. The RemoveNthFrom method is called by this macro:

```
int32 RemoveNthFromObject( CObject *obj, int32 N )
```

See "Removing Objects From a Collection" on page 219 for more information.

Start

The Start method, called by the macro StartObject(), works the same for a collection as it does for a sequence, that is, it makes a collection active so that it can be played by the Juggler.

When a collection is started, all of its component objects are started with it, and play back in parallel (not in sequence, as the events within a sequence play).

A collection, like a sequence, stores Time and NumReps values in the appropriate variables so the Juggler can use those values to determine absolute event times and how many times to repeat the collection. During repeats, all of the collection's constituent objects are replayed.

Stop

The Stop method, called by the macro StopObject(), works the same for a collection as it does for a sequence; that is, the collection is made inactive so that it is no longer be played by the Juggler. When a task stops a collection, the collection sends Stop messages to all of its constituent objects so that they stop as well.

Note: A task should not stop a collection that is a constituent of a higher collection, because the results are unpredictable. The higher collection may restart the collection that was stopped.

Print

This method, called by the macro PrintObject(), prints debugging information as it does for a sequence. It also prints a list of the objects contained in the collection's object list.

Playing Sequences and Collections

Sequences and collections can be created directly, using `CreateObject()` calls and methods to set the objects' characteristics or the MIDI score interpretation tools described in Chapter 11, "Playing MIDI Scores," can be used. After the Juggler objects have been created, playing them with the Juggler is a simple process. This section describes the process used to play objects with the Juggler.

A Juggler Operational Overview

The Juggler can be thought of as a person sitting in a room with a list of Juggler objects, a telephone, and no clock. The Juggler's job is to give the current time to the objects on its list so that each object can play the appropriate events. The Juggler must also tell anyone on the other end of the phone what time the next event will be executed.

Absolute and Relative Event Times

Each Juggler object contains events, whether it is a sequence with its own event list, or a collection that contains sequences as constituents. Each event comes with a timestamp that gives the relative time in ticks for the start of each event. Relative time is reckoned as the number of ticks after starting time: 30 ticks after playback starts, 486 ticks after playback starts, 1090 ticks after playback starts, and so on. These relative event times are converted to absolute event times when an object becomes active.

When an object is activated with a Start message, the message provides an absolute starting time, which the object uses to determine subsequent times for all events. This absolute starting time is usually read from the audio clock when the object is started, but can be any arbitrary value in ticks. For example, an object has three events with timestamps at the relative times 0, 30, and 600 ticks. The object is started with a Start message that provides an absolute starting time of 20,000. The object executes its events at the absolute times 34,765 ($34,765+0$), 34,795 ($34,765+30$), and 35,365 ($34,765+300$).

If the object had a start delay specified when it was started (using the tag argument `JGLR_TAG_START_DELAY`), the specified delay ticks are added to the start time, effectively delaying all absolute event times by that amount. If the start delay was 300 ticks and the provided start time was 34,765, then the delayed start time would be 35,065 ($34,765+300$). The absolute event times would then be 35,065 and 35,095 and 35,695.

Current Time

After an object is activated, the object appears in the Juggler's list of active objects so the Juggler can call it with timing information. The Juggler must get *its* timing information from an outside source. The time can be thought of as coming from outside phone calls because there is no clock available to the Juggler.

The time call comes from the task that runs the Juggler, a process called *bumping the Juggler*. Each call indicates the time in ticks, such as 20,045 or 34,765 or something similar. The calling task usually gets its time values from the audio timer, which provides a steady measure of time. However, the task can come up with time values from any source, such as vertical blanks. Or it can, if it wants, make up completely arbitrary time values to give to the Juggler, a convenient testing technique, as shown in the sample code in "An Example Program" on page 228.

Event Execution

When the Juggler receives a current time value from a calling task (referred to as bumping the Juggler), the Juggler passes that value on to each of the objects in its active object list. Each object then looks to see if it has any unexecuted events that should have occurred by the current time value received. If it finds unexecuted events, it immediately executes them and marks them as executed.

If the calling task sends time values infrequently, consecutive events in an event list can be played simultaneously. For example, if a sequence has an event list with events occurring every 30 ticks and the calling task provides current time values 300 ticks apart, the object executes ten simultaneous events each time it receives a time value.

Time Intervals

To keep events from executing simultaneously and to keep time values flowing smoothly, the calling task could send a new time value to the Juggler once every tick. This is a bad idea, though, similar to busy waiting, and can prevent other tasks from sharing system resources with your task. The Juggler returns the absolute time of the next event to the calling task, which can use that time to determine when to send the next time value.

When each active object receives the current time value from the Juggler, it executes the events that need execution and also looks to see what events are yet to be executed. A sequence finds the next event following the current time and reports the execution time of that event. A collection gathers the next event execution times from all of its constituent objects and reports the execution time of the earliest event. The Juggler gathers the next event execution times from all the active objects in its list and reports the earliest time back to the calling task.

When the calling task receives the absolute time of the next event, it can put itself into wait state or go on to do other things until that time occurs. When the time comes, the task calls the Juggler with the current time. The Juggler informs all the active objects of the time; they execute the event (or events) that need execution and report the time of the next event, which the Juggler returns to the calling task. The task waits again for the next event time and repeats this cycle until there are no more events to execute.

The Juggler Process

To use the Juggler for event execution, a task usually follows these steps:

- ◆ Initialize the Juggler.
- ◆ Create a score of events by creating the appropriate sequences and collections.
- ◆ Poll the audio timer for the current time.
- ◆ Use the current time to start all the objects to be played.
- ◆ Send the current time to the Juggler (referred to as bumping the Juggler).
- ◆ Retrieve the next event time returned by the Juggler.
- ◆ Wait until the next event time.
- ◆ Repeat the last three steps until there are no more events to execute.
- ◆ Stop the objects.
- ◆ Destroy the objects if they are no longer needed.
- ◆ Terminate the Juggler.

Bumping the Juggler

To bump the Juggler, a task uses the following call:

```
int32 BumpJuggler( Time CurrentTime, Time *NextTime,  
                  int32 CurrentSignals, int32 *NextSignals )
```

The call accepts four arguments: `CurrentTime`, the current time value supplied by the calling task; `*NextTime`, a pointer to a time variable where the Juggler stores the next event time; `CurrentSignals`, a signal mask containing the current signals received; and `*NextSignals`, a pointer to a signal mask variable where the Juggler stores the next event signal.

Note: *The current version of the Juggler does not support signal designated events. The `CurrentSignals` argument should be set to 0 and the `*NextSignals` argument should point to a dummy variable.*

A task making this call usually gets the current time value from the audio timer, but can supply whatever arbitrary value it wishes.

When executed, `BumpJuggler()` passes the `CurrentTime` value to the Juggler, which passes it on to the active objects in its list. The objects then execute unexecuted events that should have been executed by the supplied time. `BumpJuggler()` stores the time of the next event to be executed in the `NextEvent` variable, where the calling task can retrieve the time.

`BumpJuggler()` returns a 1 if every object is finished playing or if there are no active objects. It returns a 0 if it was successful and it executed events in active objects. It returns a negative value (an error code) if unsuccessful.

Terminating the Juggler

When a task finishes using the Juggler to manage events and Juggler objects, the task should terminate the Juggler using this call:

```
int32 TermJuggler( void )
```

This call accepts no arguments and, when executed, terminates the Juggler and frees any resources used to support the Juggler and Juggler objects.

An Example Program

Example 12-7 shows how a task uses the Juggler to handle events in a collection with two constituent sequences. To keep the program simple, the events are not audio events, the interpreter procedure for each sequence is a print function that gives information about the time of execution and the data contained in the event. To further simplify the program, the task does not consult the audio timer for the current time, but simply generates the current time itself, first using an initialized value, then using the previous next event time as the current time.

Example 12-7 *An example program.*

```
/* *****  
**  
** Test Juggler using dumb non-musical events and software "timing"  
** Construct a hierarchy of a Parallel Collection and two Sequences  
**  
** By:   Phil Burk  
**  
** Copyright (c) 1993, 3DO Company.  
** This program is proprietary and confidential.  
**  
***** */  
  
#include "types.h"  
#include "debug.h"  
#include "nodes.h"
```



```

#include "list.h"
#include "stdarg.h"
#include "strings.h"
#include "operror.h"
#include "strings.h"
#include "stdio.h"
#include "audio.h"
#include "music.h"

#define PRT(x){ printf x; }
#define ERR(x) PRT(x)
#define DEBUG(x) /* PRT(x) */

/* Macro to simplify error checking. */
#define CHECKPTR(ptr,name) \
    if (ptr == NULL) \
    { \
        ERR(("Failure in %s\n", name)); \
        goto cleanup; \
    }

/*****
/* Define our basic event type. Each sequence could be different. */
typedef struct
{
    Time    te_TimeStamp;
    uint32 te_Data;
} TestEvent;

/* This function is called when the Sequence repeats. */
int32 UserRepeatFunc ( Jugglee *Self, Time RepeatTime )
{
    PRT(("====Repeat function for 0x%x at Time %d\n", Self, RepeatTime));
    return 0;
}

/* This function is called by the Sequence to interpret the current event. */
int32 UserInterpFunc ( Jugglee *Self, TestEvent *te )
{
    PRT(("TestEvent: Self = 0x%x, Time = %d, Data = %d\n",
        Self, te->te_TimeStamp, te->te_Data));
    return 0;
}

TestEvent TestData[] =
{

```

```
{ 5, 567 },
{ 7, 910 },
{ 12, 400 }
};

#define STARTTIME (2000)
int main()
{
    Sequence *seq1 = NULL;
    Sequence *seq2 = NULL;
    Collection *col = NULL;
    TagArg Tags[8];
    int32 Result;
    Time CurTime, NextTime;
    int32 NextSignals;

    /* Initialize audio, return if error. */
    /* This is required in early versions of the Juggler.
       It should not be required later. */
    if (OpenAudioFolio())
    {
        ERR(("Audio Folio could not be opened!\n"));
        return(-1);
    }

    PRT(("Test Objects\n"));
    InitJuggler();

    /* Instantiate objects from class. */
    seq1 = (Sequence *) CreateObject( &SequenceClass );
    CHECKPTR(seq1, "CreateObject" );

    seq2 = (Sequence *) CreateObject( &SequenceClass );
    CHECKPTR(seq2, "CreateObject" );

    col = (Collection *) CreateObject( &CollectionClass );
    CHECKPTR(col, "CreateObject" );

    /* define TagList */
    Tags[0].ta_Tag = JGLR_TAG_REPEAT_FUNCTION;
    Tags[0].ta_Arg = (void *) UserRepeatFunc;
    Tags[1].ta_Tag = JGLR_TAG_INTERPRETER_FUNCTION;
    Tags[1].ta_Arg = (void *) UserInterpFunc;
    Tags[2].ta_Tag = JGLR_TAG_MAX;
    Tags[2].ta_Arg = (void *) 3;
    Tags[3].ta_Tag = JGLR_TAG_MANY;
    Tags[3].ta_Arg = (void *) 3;
```

```
Tags[4].ta_Tag = JGLR_TAG_EVENTS;
Tags[4].ta_Arg = (void *) &TestData[0];
Tags[5].ta_Tag = JGLR_TAG_EVENT_SIZE;
Tags[5].ta_Arg = (void *) sizeof(TestEvent);
Tags[6].ta_Tag = JGLR_TAG_START_DELAY;
Tags[6].ta_Arg = (void *) 2;
Tags[7].ta_Tag = TAG_END;

/* Set various parameters in object by using TagList */
SetObjectInfo(seq1, Tags);
Tags[6].ta_Tag = JGLR_TAG_START_DELAY;
Tags[6].ta_Arg = (void *) 6;
SetObjectInfo(seq2, Tags);

PrintObject( seq1 );
PrintObject( seq2 );

/* Add Sequences to Collection for parallel play */
Result = col->Class->Add(col, seq1, 3);
if (Result)
{
    PRT(("Add returned 0x%x\n", Result));
}
Result = col->Class->Add(col, seq2, 4);
if (Result)
{
    PRT(("Add returned 0x%x\n", Result));
}

/* Start Collection which starts both Sequences. */
Result = StartObject(col, STARTTIME, 3, NULL);
if (Result)
{
    PRT(("Start returned 0x%x\n", Result));
}

/* Drive Juggler using fake time. */
NextTime = STARTTIME - 2;
do
{
    CurTime = NextTime;
    PRT(("CurTime = %d\n", CurTime ));
    Result = BumpJuggler( CurTime, &NextTime, 0, &NextSignals );

} while ( Result == 0);
```

```
StopObject(seq1, NULL);
StopObject(seq2, NULL);

cleanup:
DestroyObject( (COBObject *) seq1 );
DestroyObject( (COBObject *) seq2 );
DestroyObject( (COBObject *) col );

TermJuggler();
}
```

Function Calls and Method Macros

The following function calls and method macros are used to control the Juggler and manage Juggler objects. See Chapter 2, "Music Library Calls" in the *3DO Music and Audio Programmer's Reference* for additional information on these calls.

Juggler Control Calls

The following calls control the Juggler:

BumpJuggler()	Bumps the Juggler data-structure index.
InitJuggler()	Initializes the Juggler mechanism for controlling events.
TermJuggler()	Terminates the Juggler mechanism for controlling events.

Object Management Calls

The following calls manage Juggler objects:

<code>CreateObject()</code>	Creates an object of the given class.
<code>DefineClass()</code>	Defines a class of objects.
<code>DestroyObject()</code>	Destroys an object.
<code>ValidateObject()</code>	Validates an object.

Method Macros

The following macros manage Juggler objects:

<code>AllocObject()</code>	Allocates memory for an object.
<code>FreeObject()</code>	Frees memory allocated for an object.
<code>GetObjectInfo()</code>	Gets the current settings of an object.
<code>GetNthFromObject()</code>	Gets the nth element of a collection.
<code>PrintObject()</code>	Prints debugging information about an object.
<code>RemoveNthFromObject()</code>	Removes the nth element of a collection.
<code>SetObjectInfo()</code>	Sets values in the object based on tag arguments.
<code>StartObject()</code>	Starts an object so the Juggler will play it.
<code>StopObject()</code>	Stops an object so the Juggler will not play it.

Object-Defining Tag Arguments

The following tag arguments can be used to set Juggler object variables:

```
JGLR_TAG_EVENTS
JGLR_TAG_EVENT_SIZE
JGLR_TAG_MAX
JGLR_TAG_MANY
JGLR_TAG_INTERPRETOR_FUNCTION
JGLR_START_FUNCTION
JGLR_TAG_REPEAT_FUNCTION
```

JGLR_TAG_STOP_FUNCTION
JGLR_TAG_START_DELAY
JGLR_TAG_REPEAT_DELAY
JGLR_TAG_STOP_DELAY
JGLR_TAG_DURATION
JGLR_TAG_SELECTOR_FUNCTION
JGLR_TAG_MUTE
JGLR_TAG_CONTEXT

Tips and Techniques

This chapter provides programming tips and techniques about audio programming and the Music library. This chapter contains:

- ◆ Answers to questions developers ask on the 3DO InfoServer bulletin board
- ◆ Answers to questions that the Developer Support Group received
- ◆ Information on preparing audio, using scores, voices, and instruments, working with timing issues, and playing sound files
- ◆ Troubleshooting advice

This chapter contains the following topics:

Topic	Page Number
DSP Resources	
Managing Sound Effects	
Attaching Multiple Samples to an Instrument	
Audio Samples	
Playing Scores	
Playing Red Book Audio	
Timing	
Troubleshooting	
Filters	
Miscellaneous	

DSP Resources

Dynamic Voice Allocation

Q: Is dynamic voice allocation supported?

A: The score playing code in the Music library supports dynamic voice allocation. When a voice is needed, the code attempts the following:

- ◆ Adopt an unused instrument of the same type.
- ◆ If that fails create a new instrument.
- ◆ If that fails adopt another instrument of a different type, kill it and use its resources.
- ◆ If that fails scavenge instruments from itself or others.

The code uses the Audio folio calls `AbandonInstrument()`, `AdoptInstrument()`, `ScavengeInstrument()`, and `GetAudioItemInfo()` to implement these functions.

For an example of using the score player as a sound effects manager, see the example program *sfx_score.c*.

Audio Samples

Available RAM for Sampling

Q:How much RAM is available for sampled instruments?

A:You can use any of the memory for audio samples; but you have to decide how much you want to commit to audio. The technical limit of the address bus is 16 MB.

Looping Stereo and Mono Sound

Q:How well do AIFF files loop?

A:Loops can be set at any byte boundary, so stereo 16-bit data can loop at any frame. It does not matter whether the sample is streaming off the disc from a sound file or is in memory.

Translating PC VOX files

Q: Can I translate PC VOX files to AIFF files with SoundHack?

A: No. SoundHack can convert between 10 different formats, including IRCAM format, but PC VOX is not supported. The new SoundHack V0.70 does, however, support Microsoft WAV format files. If you convert from PC VOX to WAV using some PC based tool, you could convert from WAV to AIFF using SoundHack

V0.70. Another alternative is to convert your files to RAW binary data files, then read them using SoundHack, set the header info to 8-bit, 22,050 Hertz, then write them out as AIFF files.

Loading Multiple Sound Files

Q: My game has dozens of digitized sound files that I must frequently load from the CD-ROM and subsequently unload to make space for something else.

Although the CD-ROM loads a fairly adequate speed, the seek-to-read time is very long. Loading ten digitized sound files from CD-ROM at the beginning of a screen takes a long time.

- ◆ I need a way to load multiple concatenated digitized sound files at once to get around the seek time. This would allow us to manage our memory better.
- ◆ After long runs I seem to experience memory fragmentation problems, and so would like to be able to supply to the sample loader routines the address of a buffer to use. Memory does not have a chance to fragment if the same buffer is reused each time.

A: You do not need to load files in AIFF format. You can load them using your own optimized loader, then make them available to the Audio folio by calling `CreateSample()`. You can then determine their placement in memory any way you choose.

Playing Scores

Streaming Audio versus Score Playback

Q: How are most artists implementing sound: using streamed CD music or by using MIDI?

A: The artists look at the advantages of MIDI versus streamed music for their specific application. The advantages of MIDI are as follows:

- ◆ More interactive. You can tweak the sequence, mute tracks, and so on
- ◆ Leaves the disc free for other accesses
- ◆ Can incorporate algorithmic elements for infinite variety
- ◆ Takes less disc space than streaming

On the other hand streamed audio can

- ◆ Play any music with vocals, animal sounds, whatever
- ◆ Leave DSP pretty much free for sound effects
- ◆ Require less RAM than MIDI

Score Files

Q: How can I create a 3DO-compatible score file?

A: The Music library score playing code reads standard MIDI files as input. You can create MIDI files with almost any commercial sequencer or music notation program. You then need to create a PIMap file that associates MIDI program numbers with 3DO instruments or samples. These are text files that can be created with the MPW editor. They consist of lines with program numbers followed by sample or instrument filenames. (See Chapter 2, "Music Library Calls," for more details.)

You can create samples using AudioMedia. Since AIFF is a standard format, there are many sources for sample files. The 3DO content library is a good source. We are also contracting for the development of 3DO instrument libraries.

The ARIA tool helps you edit PIMaps and play scores.

Q: How can I play my score on the 3DO Interactive Multiplayer?

A: Use the *playmf* example program, which accepts a MIDI filename and a PIMap filename as input. You can adapt the source code included in the release to your application.

Q: Can I control scores once they're playing?

A: Yes. The score should be as tightly coupled to application state as possible. You can make Juggler calls to mute various tracks as they play and to select alternative melodies based on application state. It is also possible to edit sequences while they are playing and to write custom score interpretation functions that are used by the Juggler for special purposes.

Q: What about simultaneous score play?

A: Simultaneous score play should be transparent under multi-tasking. Just be careful of the limits of the DAC amplitude, which is a maximum of 1.0 per left or right channel.

Q: How do scores interact with sound effects?

A: Scores are just sequences of arbitrary events which could be notes, sound effects, changes to sound effects currently running, graphics, or whatever. You can call `StartScoreNote()` to trigger sound effects within the score context. Then sound effects and notes from the score are dynamically allocated from the same pool. You also have full access to the Audio folio when playing scores.

MIDI

Q: Can I use MIDI?

A: You can play MIDI on the 3DO Station from inside your program. The ARIA tool also makes real-time playback of MIDI files possible. Make the output of your favorite MIDI sequencer the input to ARIA, then play your MIDI files and hear them directly on the 3DO Station.

Q: Is it possible to modify the characteristics of a MIDI PIMaps instrument. In particular I would like to set up a PIMap entry and somehow connect other instruments to that entry and tweak the knobs.

A: The problem is that the contents of the PIMap are Templates, not Instruments. The score player allocates instruments on the fly as required to play the score. It would be hard to intercept those and connect other instruments to them.

Q: Can MIDI File tracks be polyphonic?

A: Yes. Each individual MIDI File track can contain notes on up to 16 MIDI channels. On any given channel, the maximum number of voices is determined by which program sound is playing on that channel. The maximum number of voices for a program is set using the `-m` option in the PIMap file, or as a parameter to:

```
SetPIMapEntry(scon, ProgramNum, InsTemplate, MaxVoices, Priority) .
```

In a PIMap, for example:

```
7 flute.aiff -m 4 ; allows up to 4 part polyphony on the flute
```

Playing Red Book Audio

Q: How can I play Red Book audio as part of my title?

A: You cannot interleave the proprietary 3DO format with Redbook Audio. Convert the sounds to an AIFF file and use the Sound Player or the 3DO Data Streamer. You can extract sound from a Redbook CD using commercially available programs such as Disc-to-Disk. If you have sound files in other formats, you can convert them using SoundHack.

Timing

DSP Time versus System Time

Q: What is the link between DSP time and system time?

A: The audio/music timer is divided down from the DSP frame rate of 44.1 kHz. The microsecond or the vertical blanking timer can also be used but are not synchronous with the audio clock.

Multi-thread Timers

Q: I have some success with `GetAudioTime` using ticks. I was not sure if multiple threads could use the timer at the same time?

A: Yes they can. The restriction is that cue items must be allocated by the thread that uses them.

Filters

Modulating Selected Frequency Ranges

Q: Is there a way to amplify a specific range of frequencies during the play of an instrument? This is equivalent to the function of a graphic equalizer on a boom box. I am specifically interested in increasing the amplitude in the "bass" range.

A: Yes. You can use the `svfilter.dsp`. In addition to the Output, which is a lowpass filter, it also has BandPass and HighPass outputs. You can use the BandPass output or the LowPass, optionally mixed with original signal, to boost bass response.

Another approach is to use the EQ or filter effects in Sound Designer to boost the bass in the sample before you play it.

Miscellaneous

Allocating Signals

Q: I have two tasks, A and B. I want to have task A send a signal to task B using `SendSignal()` and I want task B to receive the signal using `WaitSignal()`. Which task allocates the signal, A the sender, or B the receiver?

A: B, the receiver.

Beep Folio

This chapter describes the Beep Folio which is a new audio feature of the 3DO M2 Portfolio. Since Beep Folio is an alternative to the Audio Folio, we start by describing the differences between Beep Folio and Audio Folio and the reasons for choosing each one. This chapter also provides a Beep Folio vocabulary of concepts and step-by-step instructions for using the new folio.

This chapter contains the following topics:

Topic	Page Number
Differences Between Beep Folio and Audio Folio	219
Beep Folio Vocabulary	222
Steps to Use the Beep Folio	222
Other Considerations	223

Differences Between Beep Folio and Audio Folio

Beep Folio provides access to the 3DO audio hardware at a much simpler level than Audio Folio and because it provides less functionality than the Audio Folio it has less overhead. Beep folio is less well supported than Audio Folio and it is more complicated to use. If you are low on memory, or have simple requirements, Beep Folio may work better for you than Audio Folio.

The Beep Folio provides a mechanism for loading a single DSP program that has multiple voices. This program can be considered as a virtual machine and is called a *Beep Machine*. Many hardware platforms do not have a DSP. They have a non-programmable, fixed hardware architecture for audio. The Beep Folio mimics those systems by providing one or more monolithic Beep Machines.

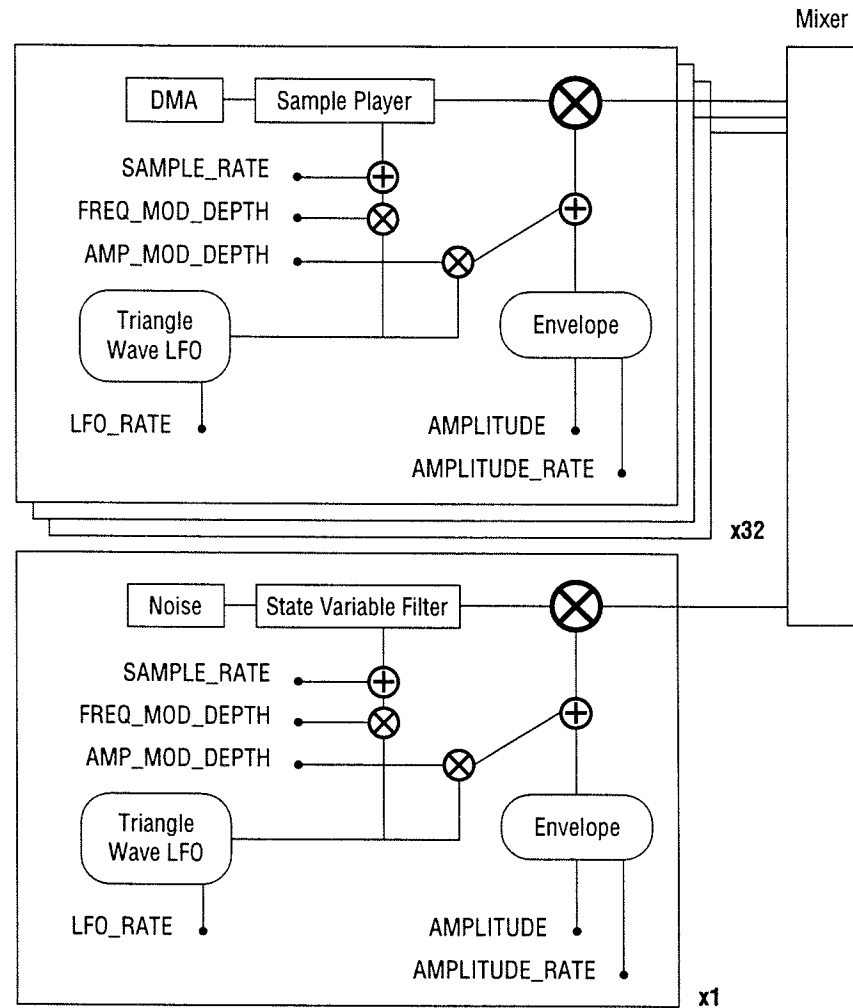


Figure 13-1 *Beep Machine*

The basic machine provided in the 2.0 release offers 32 voices of sampled sound playback as well as a filtered noise effect. The amplitude of each voice is controlled by an envelope. You can use the envelope to change the amplitude smoothly thus avoiding the loud pop that can occur with sudden changes in amplitude.

In contrast, the Audio Folio provides a mechanism for loading a variety of DSP instruments and connecting them together in real time. Thus the Audio Folio

provides a much more powerful audio synthesis environment. This power requires extra memory, however, and may not be needed by all applications.

Reasons for Choosing Audio Folio

If you answer *Yes* to any of the following questions, then you must use the Audio Folio:

Does your application require DSP instruments that are not available in a Beep Machine such as `rednoise.dsp` or `schmidt_trigger.dsp`?

Does your application require the Audio Patch Folio for generating complex patches? The Patch Folio uses the Audio Folio Template Items and will not work with the Beep Folio.

Does your application require any of the following Music Library functions: 3D Sound, MIDI Score Playing, Sound Spooler, or Advanced Sound Player for AIFF?

Does your application use the Data Streamer for playing movies?

Reasons for Choosing Beep Folio

If you answer *Yes* to all of the following questions, then you might consider using the Beep Folio:

Is your application desperately low on memory. The Beep Folio will save you about 80K bytes of main memory relative to the Audio Folio.

Are your audio needs very simple and mostly limited to simple playback of audio samples?

Are you willing to do some low level work normally handled by the Audio Folio like analyzing audio sample information and carefully controlling audio DMA in order to play them?

What Beep Folio Doesn't Provide

The ability to load an arbitrary mixture of instruments.

The ability to connect simple instruments together to form a complex patch.

The ability to share the DSP between applications.

The ability to link attachments together to simplify spooling.

Support for ARIA instruments.

Beep Folio Vocabulary

The Beep Folio literature uses a number of familiar words in different ways. The most important words and the ways that they are used in describing the Beep Folio are listed and explained here.

Machine

A large DSP program that can generate multiple voices. The Beep Folio can load one of these at a time.

Voice

A sound source that can be controlled independently. A Beep Machine typically has multiple voices of different types including sampled sound players and filtered noise. Voices are specified by index.

Channel

A DMA channel that can move audio data from memory into the DSP. Each sampled sound voice has a DMA channel associated with it. Voices that do not play samples will have no associated DMA channel. Voices with DMA channels are assigned first so that the channel associated with a voice has the same index as the voice.

Parameter

A controllable value associated with a Beep Machine or a single voice in that machine. Parameters are identified by constants defined in the include file associated with a given Beep Machine. These constants contain information used by the Beep Folio to convert the given floating point value to the appropriate internal DSP value.

Steps in Using the Beep Folio

1. Decide which Beep Machine to use and `#include` that machine's include file in your application. (At the current time there is only one Beep Machine available so this is an easy decision.
2. Call `OpenBeepFolio()` to dynamically link with the folio.
3. Call `LoadBeepMachine(BEEP_MACHINE_NAME)` to load the machine program into the DSP.
4. Optionally set the `BMVP_LEFT_GAIN` and `BMVP_RIGHT_GAIN` parameters to control the mixing of your sounds.
5. Load any samples that you wish to play into memory using `GetAIFFSampleInfo()` or another method of your own design.

6. Configure the DMA channels for the appropriate sample data format, address and length using `ConfigureBeepChannel()`, `SetBeepChannelData()` and `SetBeepChannelDataNext()`.
7. Set any parameters specific to your beep machine that you wish using `SetBeepVoiceParameter()` and `SetBeepParameter()`.
8. Start various DMA channels using `StartBeepChannel()` to play samples.
9. Stop the DMA using `StopBeepChannel()`
10. Finish by calling `UnloadBeepMachine()` and `CloseBeepFolio()`.

Other Considerations

What follows here are a number of tips to make your life with Beep Folio easier.

Mixing

Beep Machines contain an internal mixer that combines the output of the different voices. Each voice has a `BMVP_LEFT_GAIN` and a `BMVP_RIGHT_GAIN` that determines how much of each voice goes into the global mix. The default setting for these typically allow several voices to be played simultaneously without clipping. If you hear your output clipping then you should turn down these gain parameters. Clipping sounds like a harsh raspy noise whenever the original sound is loud. If your sounds are too quiet then turn up these gains. The best way to set the levels is to play an audio CD on your 3DO dev station and set your amplifier to a comfortable level. Then adjust the mixer gains in your title so that you can play it on the same system without having to touch the volume knob on your amplifier.

Cache Coherency

Caution: If you write to your sample data memory using the CPU, then you must flush the data cache before playing that sample. Otherwise the data may still be sitting in the data cache when the audio DMA tries to read it. This could result in pops or unexpected sound bursts. You can flush the data cache using `WriteBackDCache()`.

Knowing when a sample is finished playing

You can use the signal parameter in `SetBeepChannelDataNext()` to tell you when a sample has finished playing. The signal parameter is normally used to tell you when the DMA hardware has started playing the Next data. If you want to know when a sample is finished, setup a short section of silence to play at the end of your sample. When the DMA starts playing the silence you will get back your signal and you will know the sample has finished.

Index

A

absolute event times MPG-211

advanced sound player

- adding sound files MPG-119

- cleanup MPG-120

- creating MPG-119

- features MPG-118

- looping and branching MPG-121

- setting branches MPG-121

- start playing sound MPG-119

- start reading sound MPG-119

advanced sound player example MPG-118

AF_TAG_ADDRESS MPG-48, MPG-84

AF_TAG_AMPLITUDE MPG-36

AF_TAG_BASEFREQ MPG-48

AF_TAG_BASENOTE MPG-48

AF_TAG_CHANNEL MPG-48

AF_TAG_CLEAR_FLAGS MPG-60, MPG-84

AF_TAG_COMPRESSIONRATIO MPG-48

AF_TAG_COMPRESSIONTYPE MPG-48

AF_TAG_DELAY_LINE MPG-48

AF_TAG_DETUNE MPG-48

AF_TAG_FRAMES MPG-48, MPG-85

AF_TAG_HIGHNOTE MPG-48

AF_TAG_HIGHVELOCITY MPG-48

AF_TAG_HOOKNAME MPG-60

AF_TAG_INSTRUMENT MPG-60

AF_TAG_LOWNOTE MPG-48

AF_TAG_LOWVELOCITY MPG-49

AF_TAG_MAX MPG-80

AF_TAG_MIN MPG-80

AF_TAG_NAME MPG-81

AF_TAG_NUMBITS MPG-49

AF_TAG_NUMBYTES MPG-49

AF_TAG_PITCH MPG-35

AF_TAG_RATE MPG-35

AF_TAG_RELEASEBEGIN MPG-49, MPG-85

AF_TAG_RELEASEEND MPG-49, MPG-85

AF_TAG_RELEASEJUMP MPG-85

AF_TAG_RELEASETIME MPG-85

AF_TAG_SAMPLE MPG-60

AF_TAG_SAMPLE_RATE MPG-49

AF_TAG_SET_FLAGS MPG-60, MPG-85

AF_TAG_START_AT MPG-60

AF_TAG_SUSTAINBEGIN MPG-49, MPG-85

AF_TAG_SUSTAINEND MPG-49, MPG-85

AF_TAG_SUSTAINTIME MPG-85

AF_TAG_TIME_SCALE MPG-36, MPG-60

AF_TAG_VELOCITY MPG-36

AF_TAG_WIDTH MPG-49

AIFC MPG-46

AIFC format MPG-24

AIFF MPG-46

AllocObject() MPG-206

amplitude value

- setting up MPG-156

applying tag argument values MPG-203

Aria MPG-22

attachment

- releasing independent MPG-63

- stopping independent MPG-63

attachment item MPG-51

attachments MPG-20

- attributes MPG-59, MPG-60

- independent MPG-62

- instrument-stopping MPG-62

- linking MPG-63

Numerics

3D Sound MPG-117

A

A MPG-94

absolute event times MPG-203

advanced sound player

adding sound files MPG-127

cleanup MPG-127

creating MPG-127

features MPG-126

looping and branching MPG-129

setting branches MPG-129

start playing sound MPG-127

start reading sound MPG-127

advanced sound player example MPG-126

AF MPG-80

AF_ENVF_LOCKTIMESCALE MPG-79, MPG-82

AF_TAG_ADDRESS MPG-49, MPG-75

AF_TAG_AMPLITUDE_FP MPG-36

AF_TAG_BASEFREQ MPG-49

AF_TAG_BASENOTE MPG-49

AF_TAG_CHANNEL MPG-49

AF_TAG_CLEAR_FLAGS MPG-52, MPG-82

AF_TAG_COMPRESSIONRATIO MPG-49

AF_TAG_COMPRESSIONTYPE MPG-49

AF_TAG_DELAY_LINE MPG-49

AF_TAG_DETUNE MPG-49

AF_TAG_FRAMES MPG-49, MPG-76

AF_TAG_HIGHNOTE MPG-49

AF_TAG_HIGHVELOCITY MPG-49

AF_TAG_HOOKNAME MPG-52

AF_TAG_INSTRUMENT MPG-52

AF_TAG_LOWNOTE MPG-49

AF_TAG_LOWVELOCITY MPG-49

AF_TAG_MAX MPG-70

AF_TAG_MIN MPG-70

AF_TAG_NAME MPG-70

AF_TAG_NUMBITS MPG-50

AF_TAG_NUMBYTES MPG-50

AF_TAG_PITCH MPG-36

AF_TAG_RELEASEBEGIN MPG-50, MPG-77

AF_TAG_RELEASEEND MPG-50, MPG-77

AF_TAG_RELEASEJUMP MPG-77

AF_TAG_RELEASETIME MPG-77

AF_TAG_SAMPLE MPG-52

AF_TAG_SAMPLE_RATE MPG-50

AF_TAG_SET_FLAGS MPG-52, MPG-82

AF_TAG_START_AT MPG-52

AF_TAG_SUSTAINBEGIN MPG-50, MPG-76

AF_TAG_SUSTAINEND MPG-50, MPG-76

AF_TAG_SUSTAINTIME MPG-76

AF_TAG_VELOCITY MPG-37

AF_TAG_WIDTH MPG-50

AIFC MPG-46

AIFC format MPG-45

AIFF MPG-46

AllocObject() MPG-198

amplitude value

setting up MPG-164

applying tag argument values MPG-195

attachment

releasing independent MPG-57

stopping independent MPG-57

attachment item MPG-53

attachments MPG-20

attributes MPG-55

independent MPG-57

instrument-stopping MPG-56

linking MPG-58

modifying attributes MPG-55

setting attributes MPG-55

start independent MPG-56

starting dependent MPG-34

starting independent MPG-57

starting point for reverberations MPG-98

stopping MPG-35

tag args MPG-55

attributes

audio MPG-88

reading MPG-88

setting MPG-88

Audio MPG-75

audio

Red Book MPG-217

audio attributes MPG-88

reading MPG-88

setting MPG-88

audio clock MPG-40

checking current setting MPG-41

audio clock rate MPG-168

setting MPG-168

audio clock speed

setting MPG-157

Audio folio MPG-10, MPG-19
 closing MPG-39
 opening MPG-21
 audio hardware MPG-9
 audio sample
 attaching to instrument MPG-2
 playing MPG-2
 audio samples
 looping MPG-214
 audio signal MPG-61
 audio software MPG-10
 writing MPG-1
 audio ticks MPG-40
 audio timer
 for use with tick values MPG-189

B

Beep Folio MPG-219
 bend value MPG-93
 bending a pitch MPG-176
 BendInstrumentPitch() MPG-93, MPG-177
 Binary MPG-103
 bumping the Juggler MPG-204
 BumpJuggler() MPG-157, MPG-158, MPG-205

C

calling methods directly MPG-198
 CD player MPG-8
 ChangeScoreControl() MPG-167, MPG-175
 ChangeScorePitchBend() MPG-168, MPG-177
 ChangeScoreProgram() MPG-167, MPG-175
 changing a program MPG-175
 channel messages MPG-152
 class MPG-185
 defining MPG-187
 CloseAudioFolio() MPG-180
 CloseMathFolio() MPG-180
 collection
 adding objects to MPG-196
 creating and instance of MPG-195
 messages for MPG-201
 removing objects from MPG-197
 setting up objects for MPG-195
 setting variables with Tag Arg values MPG-196
 collection class MPG-188, MPG-190, MPG-195

compact disc player MPG-9
 Red Book MPG-10
 Yellow Book MPG-10
 Control Messages MPG-152
 control signal instruments MPG-23, MPG-24
 Convert12TET_F16 MPG-94
 create instrument MPG-20
 CreateEnvelope() MPG-83
 CreateInstrument() MPG-28
 CreateObject() MPG-203
 CreateScoreContext() MPG-159
 CreateTuning() MPG-91
 creating a collection MPG-195
 creating a Juggler object MPG-165
 creating a MIDI environment MPG-157, MPG-158
 creating a score context MPG-159
 creating a virtual MIDI device MPG-153
 creating an object MPG-187
 creating collections MPG-190
 creating sequences MPG-190
 cue MPG-40

D

data structures MPG-181
 DebugSample() MPG-54
 decision functions MPG-130, MPG-131
 rules MPG-133
 DefineClass() MPG-187
 defining a new class MPG-187
 delay instrument MPG-94
 delay line MPG-94, MPG-95, MPG-96
 calculating MPG-109, MPG-111
 connecting MPG-96
 creating MPG-95
 deleting MPG-96
 oscilloscope data MPG-99
 delay time
 setting MPG-109
 delayed playback MPG-95
 DeleteEnvelope() MPG-85
 DeleteInstrument() MPG-38
 DeleteScoreContext() MPG-179
 DeleteTuning() MPG-92
 destroying an object MPG-187
 DestroyObject() MPG-187
 DetachSample() MPG-54, MPG-85
 digital signal processor MPG-8, MPG-9, MPG-139
 digitized audio in DRAM MPG-9

direct digital input MPG-9
DisownAudioClock() MPG-179
DMA MPG-23
doppler cue MPG-118
Drift MPG-109
DSP MPG-139
DSP frame MPG-30
DSP Resources MPG-117, MPG-118
DSP ticks MPG-30
DSP time units MPG-30
DSP timing MPG-217
dynamic voice allocation MPG-170

E

echo MPG-98
effects instruments MPG-23
envelopes MPG-14, MPG-20, MPG-74
 attaching to instrument MPG-83
 creating MPG-83
 deleting MPG-85
 detaching from instrument MPG-85
 EnvelopeSegment data structure MPG-75
 loops MPG-76
 points MPG-74
 properties MPG-74
 start point MPG-55
 starting dependent MPG-34
event execution MPG-204, MPG-205
Example MPG-42
example
 advanced sound player MPG-126
executing a MIDI message MPG-172

F

flanging MPG-109
folios
 Audio and Beep MPG-219
FreeAmplitude() MPG-179
FreeObject() MPG-199
frequency MPG-72
function calls MPG-210

G

gain
 scaling MPG-119
GetAudioDuration() MPG-41

GetAudioItemInfo() MPG-55, MPG-88
GetInstrumentPortInfoByIndex() MPG-68
GetInstrumentPortInfoByName() MPG-69
GetNthFromObject() MPG-197, MPG-202
GetNumInstrumentPorts() MPG-68
GetObjectInfo() MPG-199, MPG-200, MPG-201
GetScoreBendRange() MPG-177
grab MPG-69, MPG-89
GrabKnob() MPG-89

I

importing a MIDI score MPG-155, MPG-165
importing a MIDI score file MPG-158
initializing the Juggler MPG-186
InitJuggler() MPG-158, MPG-186
InitScoreDynamics() MPG-159
InitScoreMixer() MPG-159
instance MPG-185
instrument MPG-2
 loading MPG-2
 start playing MPG-3
 stopping MPG-6
instrument template MPG-21, MPG-39
instrument templates
 assigning MPG-155
 predefined MPG-22
InstrumentPortInfo MPG-68
Instruments MPG-22

instruments MPG-11, MPG-19
 attachment MPG-53
 connecting one to another MPG-32
 creating MPG-28, MPG-30
 deleting MPG-38, MPG-39
 disconnecting MPG-33
 elements of MPG-11
 freeing MPG-38
 knobs MPG-67
 loading MPG-20, MPG-30
 loading template MPG-26
 playing MPG-20
 preparing MPG-20
 priority MPG-12
 releasing MPG-37
 resources MPG-30
 starting MPG-34
 stopping MPG-35, MPG-38
 tag args MPG-35
 tuning MPG-90
 types MPG-22
 use of by tasks MPG-12
 using MPG-20
 interpreter procedure MPG-190
 writing MPG-192
 interpreting a MIDI message MPG-172
 InterpretMIDIEvent() MPG-157, MPG-167, MPG-172
 InterpretMIDIMessage() MPG-157, MPG-167, MPG-170, MPG-172, MPG-175

J

jack
 headphone MPG-9
 line-level output MPG-9
 JGLR_START_FUNCTION MPG-193, MPG-196
 JGLR_TAG_CONTEXT MPG-194, MPG-196
 JGLR_TAG_EVENT_SIZE MPG-193
 JGLR_TAG_EVENTS MPG-193
 JGLR_TAG_INTERPRETOR_FUNCTION MPG-193
 JGLR_TAG_MANY MPG-193
 JGLR_TAG_MAX MPG-193
 JGLR_TAG_MUTE MPG-194
 JGLR_TAG_REPEAT_DELAY MPG-194, MPG-196
 JGLR_TAG_REPEAT_FUNCTION MPG-193, MPG-196

JGLR_TAG_SELECTOR_FUNCTION MPG-194
 JGLR_TAG_START_DELAY MPG-194, MPG-196
 JGLR_TAG_STOP_DELAY MPG-194, MPG-196
 JGLR_TAG_STOP_FUNCTION MPG-194, MPG-196
 juggler class MPG-188
 Juggler MPG-183
 control calls MPG-210
 event execution process MPG-205
 example of using events MPG-206
 initializing MPG-186
 overview of MPG-203
 terminating MPG-206
 using to play collections MPG-203
 using to play sequences MPG-203
 Juggler object
 collection MPG-4
 sequence MPG-4

K

knobs MPG-11, MPG-12, MPG-20, MPG-32, MPG-33, MPG-67
 changing parameters MPG-12
 checking parameters MPG-13
 definition of MPG-12
 grab MPG-67, MPG-69, MPG-89
 parameters MPG-70
 releasing MPG-73
 tweak MPG-67

L

library functions
 importing a MIDI score file MPG-149
 LinkAttachments() MPG-58
 literal sample data MPG-46
 load instrument template MPG-20
 loading a PIMap file MPG-163
 LoadInsTemplate MPG-26
 LoadInstrument() MPG-30
 LoadPIMap() MPG-160, MPG-161, MPG-163
 LoadSample() MPG-47
 loop MPG-50, MPG-51, MPG-76

M

makepatch MPG-104
 managing objects MPG-186

- markers
 - decision functions MPG-131
 - in sounds MPG-128
 - static actions MPG-130
- MaxVoices MPG-160
- message MPG-184
- message macros MPG-198
- messages
 - for collections MPG-201
 - for sequences MPG-198
- method MPG-184
 - calling directly MPG-198
- method macros MPG-210, MPG-211
- MFDefineCollection() MPG-166
- MFLoadCollection() MPG-155, MPG-166
- MFLoadSequence() MPG-155, MPG-165
- MFUnloadCollection() MPG-179
- MIDI MPG-149
 - channels MPG-151
 - Environment Calls MPG-181
 - function calls MPG-181
 - importing a score MPG-155
 - messages MPG-151
 - playback calls MPG-182
 - program numbers MPG-160
 - providing playback functions MPG-156
 - review of MPG-151
 - Score Calls MPG-182
 - setting channels MPG-154
- MIDI environment
 - creating MPG-153
- MIDI file MPG-4
- MIDI instruments MPG-90
- MIDI messages MPG-9
- MIDI network MPG-151
- MIDI score
 - overview of playing process MPG-157
 - playing MPG-168
 - setting voice and program limits MPG-158
- MIDI score formats MPG-152
- MIDI score playback
 - initializing a mixer for MPG-164
- MIDI timing clocks MPG-152
- MIDI tuning system MPG-90
- MIDIFileParser MPG-165
- mixers MPG-20, MPG-22
- modular analog synthesizer MPG-19
- multisampling MPG-53
- Music library MPG-15, MPG-151

- musical score
 - playing MPG-4

N

- Note Off MPG-152, MPG-167
- Note On MPG-167
- NoteOffIns() MPG-174
- NoteOnIns() MPG-174
- notes
 - amplitude MPG-15
 - envelopes MPG-14
 - frequency MPG-15
 - pitch MPG-15
 - playing MPG-13
 - release MPG-14
 - sections of MPG-13
 - start MPG-14
 - stop MPG-14
 - voices MPG-15
- NotesPerOctave value MPG-91
- NoteTracker MPG-154, MPG-159

O

- object MPG-184
 - checking validity of MPG-188
 - creating MPG-187
 - destroying MPG-187
 - sending message to MPG-198
- object defining tag arguments MPG-211
- object list MPG-190
- object management calls MPG-211
- object-oriented programming MPG-184
- OpenAudioFolio() MPG-158

P

- panning MPG-118
 - delay MPG-118
- Patch MPG-103
- PatchCmd MPG-104
- PC VOX files MPG-214
- phase increment MPG-72
- PIMap MPG-155
 - directly setting entries MPG-160
 - setting entries MPG-160
 - using a MPG-161

- PIMap file MPG-4
 - loading MPG-163
 - setting up multisamples MPG-162
- pitch MPG-90, MPG-109
 - changing MPG-46
- pitch bend MPG-93
- Pitch Bend Change MPG-152
- pitch bend change messages
 - acting on MPG-177
- pitch bend value
 - creating an internal MPG-178
- pitch bend wheel MPG-176
- pitch wheel MPG-93
- pixel cornerweight MPG-xix
- placeholders MPG-190
- playback MPG-35
 - changing characteristics during MPG-178
 - creating a MIDI score for MPG-180
 - tracking MPG-155
- playback functions MPG-156
- playback tempo MPG-168
- playing process MPG-157
- playing the MIDI score MPG-168
- Preparing MPG-26
- PrintObject() MPG-201, MPG-202
- Probes
 - reading MPG-89
- Program Change MPG-152
- providing voices MPG-154

R

- Reading MPG-40
- reading a collection's object list MPG-197
- relative MPG-40
- release loop MPG-51, MPG-76
- ReleaseAttachment() MPG-57
- ReleaseInstrument() MPG-37
- ReleaseScoreNote() MPG-174
- releasing a note MPG-173
- relevant event times MPG-203
- RemoveNthFromObject() MPG-197, MPG-202
- reverb
 - MPG-111, MPG-112, MPG-113
- reverberation MPG-94
 - delay instrument MPG-94
 - delay line MPG-94

- reverberations
 - attachment starting point MPG-98
 - complex MPG-99
 - simplest MPG-111
 - submixer levels MPG-98

S

- sample
 - loading MPG-20
- sample item MPG-47
- sample size MPG-46
- sample sound pointers MPG-13
- sampled sounds MPG-125
- sampled-sound instrument
 - raw value MPG-73
- sampled-sound instruments MPG-22, MPG-23, MPG-46
 - knob MPG-73
 - mono MPG-46
 - stereo MPG-46
- samples MPG-46
 - attaching multisamples to instruments MPG-53
 - attaching to instrument MPG-51
 - attachment MPG-53
 - debugging MPG-54
 - deleting MPG-54
 - detaching MPG-54
 - FIFO MPG-52
 - frame index MPG-50
 - loading MPG-46
 - loops MPG-50
 - multisampling MPG-53
 - playback MPG-51
 - release loop MPG-51
 - sample frame MPG-50
 - simplest loading method MPG-47
 - start point MPG-55
 - starting dependent MPG-34
 - sustain loop MPG-51
 - tag args MPG-48
 - trigger points MPG-51
- score context
 - creating MPG-159
- score playback MPG-152
- ScoreChannel MPG-154
- ScoreContext MPG-155, MPG-159, MPG-177

- scores
 - files MPG-216
 - playing MPG-215
- sequence
 - setting up event list for MPG-191
 - setting variables with tag argument values MPG-193
- sequence class MPG-188, MPG-189
 - event list MPG-189
- service function MPG-127, MPG-134
- SetAudioItemInfo() MPG-88
- SetAudioRate() MPG-179
- SetObjectInfo() MPG-195, MPG-196, MPG-199
- SetPIMapEntry() MPG-160, MPG-161
- SetScoreBendRange() MPG-176, MPG-177
- setting a pitch bend range value MPG-176
- setting channel panning and volume MPG-175
- setting channels MPG-154
- setting up a MIDI score MPG-158
- setting up a mixer MPG-164
- setting up multisamples MPG-162
- setting up objects for a collection MPG-195
- sound
 - directionality MPG-119
 - environment MPG-120
 - fading smoothly MPG-6
 - producing MPG-2
- sound effects
 - complex MPG-6
 - managing MPG-5
- sound file
 - playing MPG-3
- sound files
 - loading multiple MPG-215
- sound spooler MPG-139
 - buffers MPG-134
 - convenience routines MPG-145
 - example MPG-141
 - function calls MPG-146
 - how it works MPG-140
 - how to use MPG-140
 - overview of MPG-125
 - starvation MPG-134
- sound-synthesis instrument
 - starting MPG-34
- sound-synthesis instruments MPG-22, MPG-23
- SPAction MPG-132
- spBranchatMarker() MPG-129
- spDeletePlayer() MPG-127
- specifying a user context MPG-166
- spLinkSounds() MPG-130
- spLoopSound() MPG-130
- SPMarker MPG-128
- SPPlayer MPG-128
- spService() MPG-127, MPG-134
- spSetMarkerDecisionFunction() MPG-132
- SPSound MPG-128
- square.dsp MPG-23
- ssplCreateSoundSpooler() MPG-141
- ssplDeleteSoundSpooler() MPG-141
- ssplPlayData() MPG-146
- ssplProcessSignals() MPG-141
- ssplSpoolData() MPG-141, MPG-145
- ssplStartSpooler() MPG-141
- ssplStopSpooler() MPG-141
- StartAttachment() MPG-57
- starting a note MPG-173
- starting an releasing an instrument MPG-174
- StartInstrument() MPG-34
- StartObject() MPG-200, MPG-202
- StartScoreNote() MPG-173
- static actions
 - in markers MPG-130
- StopAttachment() MPG-58
- StopInstrument() MPG-38
- StopObject() MPG-200, MPG-202
- subclass MPG-186
- submixer
 - setting levels MPG-98
- superclass MPG-186
- sustain loop MPG-51, MPG-76
- synthesizer MPG-20

T

- TagArg elements MPG-194
- TagArg values
 - applying MPG-195
- template
 - deleting MPG-39
- tempo
 - setting MPG-168
- TermJuggler() MPG-180, MPG-206
- TermScoreMixer() MPG-179
- ticks MPG-189
- timing MPG-40
 - audio ticks MPG-40

- trigger points MPG-51
 - release MPG-51
 - start MPG-51
 - stop MPG-51
- TuneInsTemplate() MPG-92
- TuneInstrument() MPG-92
- tuning MPG-90, MPG-91
 - applying to instruments MPG-92
 - creating MPG-91
 - deleting MPG-92
 - extending MPG-91
 - octaves MPG-91
- TweakKnob() MPG-71

U

- UnloadInstrument() MPG-39
- UnloadPIMap() MPG-179
- UnloadSample() MPG-54
- user context
 - specifying MPG-166
- using MIDI functions MPG-172

V

- ValidateObject() MPG-188
- virtual MIDI device MPG-153
- voice MPG-154
 - defining maximum MPG-159
- voice allocation
 - dynamic MPG-214
 - freeing instruments created by MPG-171
- voices MPG-15
 - maximum number of MPG-160



3DO M2 Audio Programmer's Reference

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Audio Programmer's Reference

1 Audio Folio Calls

--Audio Port-Types--	Audio port type descriptions.....	APR-3
--Audio Signal-Types--	Audio signal type descriptions.....	APR-4

Attachment

Create Attachment	Creates an Attachment between a Sample or Envelope to an Instrument or Template.....	APR-6
Delete Attachment	Undoes Attachment between Sample or Envelope and Instrument or Template.....	APR-8
GetAttachments	Returns list of Attachments to a sample or envelope hook.....	APR-9
GetNumAttachments	Returns the number of Attachments to a sample or envelope hook.....	APR-10
LinkAttachments.....	Connects Sample Attachments for sequential playback.....	APR-11
MonitorAttachment.....	Monitors an Attachment, sends a Cue at a specified point.....	APR-13
ReleaseAttachment.....	Releases an Attachment.....	APR-14
StartAttachment.....	Starts an Attachment.....	APR- 15
StopAttachment.....	Stops an Attachment.....	APR-16
WhereAttachment.....	Returns the current playing location of an Attachment.....	APR-17

Cue

CreateCue	Creates an audio Cue.....	APR-18
DeleteCue	Deletes an audio Cue	APR-19
GetCueSignal	Returns a signal mask of a Cue	APR-20

Envelope

CreateEnvelope	Creates an Envelope	APR-21
DeleteEnvelope	Deletes an Envelope.....	APR-22

Instrument

AbandonInstrument	Makes an Instrument available for adoption from Template.	APR-23
AdoptInstrument	Adopts an abandoned Instrument from a Template.	APR-24
BendInstrumentPitch	Bends an Instrument's output pitch up or down by a specified amount.	APR-25
ConnectInstrumentParts	Patches the output of an Instrument to the input of another Instrument.	APR-26
ConnectInstruments	Patches the output of an Instrument to the output of another Instrument.	APR-28
CreateInstrument	Allocates an Instrument from an Instrument Template.	APR-29
DeleteInstrument	Frees an Instrument allocated by CreateInstrument().	APR-31
DisconnectInstrumentParts	Breaks a connection made by ConnectInstrumentParts().	APR-32
DumpInstrumentResourceInfo	Prints out Instrument resource usage information.	APR-33
GetInstrumentPortInfoByIndex	Looks up Instrument port by index and returns port information.	APR-34
GetInstrumentPortInfoByName	Looks up Instrument port by name and returns port information.	APR-36
GetInstrumentResourceInfo	Get audio resource usage information for an Instrument.	APR-38
GetNumInstrumentPorts	Returns the number of ports of an Instrument.	APR-40
LoadInsTemplate	Loads a standard DSP Instrument \4Template\4P.	APR-41
LoadInstrument	Loads a DSP instrument \4Template\4P and creates an Instrument from it in one call.	APR-42
PauseInstrument	Pauses an Instrument's playback.	APR-44
ReleaseInstrument	Instruct an Instrument to begin to finish (Note Off).	APR-45
ResumeInstrument	Resumes playback of a paused instrument.	APR-46
StartInstrument	Begins playing an Instrument \4P (Note On).	APR-47
StopInstrument	Abruptly stops an Instrument \4P.	APR-50
UnloadInsTemplate	Unloads an instrument \4Template\4P loaded by LoadInsTemplate().	APR-51
UnloadInstrument	Unloads an instrument loaded with LoadInstrument().	APR-52

Knob

CreateKnob	Gain direct access to one of an Instrument's Knobs.	APR-53
DeleteKnob	Deletes a Knob.	APR-55
ReadKnob	Reads the current value of a single-part Knob.	APR-56
ReadKnobPart	Reads the current value of a Knob.	APR-57
SetKnob	Sets the value of a single-part Knob.	APR-58
SetKnobPart	Sets the value of a Knob.	APR-59

Miscellaneous

CloseAudioFolio	Closes the audio folio.	APR-60
EnableAudioInput	Enables or disables audio input.	APR-61
GetAudioFolioInfo	Get system-wide audio settings.	APR-62
GetAudioFrameCount	Gets count of audio frames executed.	APR-64

GetAudioItemInfo.....	Gets information about an audio item.....	APR-65
GetAudioResourceInfo.....	Get information about available audio resources.....	APR-66
OpenAudioFolio.....	Opens the audio folio.....	APR-68
SetAudioItemInfo.....	Sets parameters of an audio item.....	APR-69

Mixer

CalcMixerGainPart.....	Returns the mixer gain knob part number for a specified input and output.....	APR-70
CreateMixerTemplate.....	Creates a custom mixer <code>\f4Template\fP</code>	APR-71
DeleteMixerTemplate.....	Deletes a mixer Template created by <code>\f4CreateMixerTemplate()\fP</code>	APR-73
MakeMixerSpec.....	Makes a MixerSpec for use with <code>\f4CreateMixerTemplate()\fP</code>	APR-74
MixerSpecToFlags.....	Extracts mixer flags from MixerSpec.....	APR-76
MixerSpecToNumIn.....	Extracts number of inputs from MixerSpec.....	APR-77
MixerSpecToNumOut.....	Extracts number of outputs from MixerSpec.....	APR-78

Probe

CreateProbe.....	Creates a <code>\f4Probe\fP</code> for an output of an <code>\f4Instrument\fP</code>	APR-79
DeleteProbe.....	Deletes a <code>\f4Probe\fP</code>	APR-80
ReadProbe.....	Reads the value of a single-part <code>\f4Probe\fP</code>	APR-81
ReadProbePart.....	Reads the value of a <code>\f4Probe\fP</code>	APR-82

Sample

CreateDelayLine.....	Creates a Delay Line <code>\f4Sample\fP</code> for echoes and reverberations.....	APR-83
CreateSample.....	Creates a <code>\f4Sample\fP</code>	APR-85
DebugSample.....	Prints <code>\f4Sample\fP</code> information for debugging.....	APR-86
DeleteDelayLine.....	Deletes a delay line <code>\f4Sample\fP</code>	APR-87
DeleteSample.....	Deletes a <code>\f4Sample\fP</code>	APR-88

Signal

ConvertAudioSignalToGeneric.....	Converts audio signal value of specified signal type to generic signal value.....	APR-89
ConvertGenericToAudioSignal.....	Converts generic signal value to specified audio signal type.....	APR-90
GetAudioSignalInfo.....	Get information about audio signal types.....	APR-91

Timer

AbortTimerCue.....	Cancels a timer request enqueued with <code>\f4SignalAtAudioTime()\fP</code>	APR-93
AudioTimeLaterThan.....	Compare two AudioTime values with wraparound.....	APR-94
AudioTimeLaterThanOrEqual.....	Compare two AudioTime values with wraparound.....	APR-95
CompareAudioTimes.....	Compare two AudioTime values with wraparound.....	APR-96
CreateAudioClock.....	Creates an <code>\f4AudioClock\fP</code>	APR-97
DeleteAudioClock.....	Deletes an <code>\f4AudioClock\fP</code>	APR-98

GetAudioClockDuration	Asks for the duration of an \f4AudioClock\fP tick in frames.....	APR-99
GetAudioClockRate	Asks for \f4AudioClock\fP rate in Hertz.....	APR-100
GetAudioDuration	Asks for the duration of an AF_GLOBAL_CLOCK tick in frames.	APR-101
GetAudioTime	Reads AF_GLOBAL_CLOCK time.....	APR-102
ReadAudioClock.....	Reads AudioTime from an AudioClock.....	APR-103
SetAudioClockDuration	Changes duration of \f4AudioClock\fP tick.....	APR-104
SetAudioClockRate	Changes the rate of the \f4AudioClock\fP.....	APR-105
SignalAtAudioTime	Requests a wake-up call at a given time.....	APR-106
SignalAtTime	Requests a wake-up call at a given time from AF_GLOBAL_CLOCK.....	APR-107
SleepUntilAudioTime	Enters wait state until \f4AudioClock\fP reaches a specified AudioTime.....	APR-108
SleepUntilTime.....	Enters wait state until AF_GLOBAL_CLOCK reaches a specified AudioTime.....	APR-110

Trigger

ArmTrigger	Assigns a \f4Cue\fP to an \f4Instrument\fP's Trigger.....	APR-111
DisarmTrigger.....	Remove previously assigned \f4Cue\fP from an \f4Instrument\fP's Trigger.....	APR-113

Tuning

Convert12TET_FP	Converts a pitch bend value in semitones and cents into a float value.....	APR-114
CreateTuning	Creates a \f4Tuning\fP Item.....	APR-115
DeleteTuning.....	Deletes a \f4Tuning\fP.....	APR-117
TuneInsTemplate	Applies the specified \f4Tuning\fP to an instrument \f4Template\fP.....	APR-118
TuneInstrument	Applies the specified \f4Tuning\fP to an \f4Instrument\fP.....	APR-119

2

Audio Patch File Folio Calls

--Patch-File-Overview--	Overview of the FORM 3PCH binary patch file format.....	APR-123
EnterForm3PCH	Standard FORM 3PCH entry handler.....	APR-126
ExitForm3PCH	Standard FORM 3PCH exit handler.	APR-127
LoadPatchTemplate	Loads a binary patch file (FORM 3PCH).....	APR-129
UnloadPatchTemplate	Unloads a Patch \f4Template\fP loaded by \f4LoadPatchTemplate()\fP.....	APR-131

3

Audio Patch Folio Calls

CreatePatchTemplate	Constructs a custom Patch \f4Template\fP from simple Instrument Templates.....	APR-135
DeletePatchTemplate	Deletes a custom Instrument Template created by \f4CreatePatchTemplate()\fP	APR-137
DumpPatchCmd.....	Prints out a PatchCmd.....	APR-138

DumpPatchCmdList	Prints out a PatchCmd list.	APR-139
NextPatchCmd	Finds the next \4PatchCmd\4P in a PatchCmd list.	APR-140

PatchCmd

PatchCmd	Command set used by \4CreatePatchTemplate()\4P	APR-142
PATCH_CMD_ADD_TEMPLATE	Adds an instrument \4Template\4P to patch.	APR-143
PATCH_CMD_CONNECT	Connects patch blocks and ports to one another.	APR-144
PATCH_CMD_DEFINE_KNOB .	Defines a patch knob.	APR-146
PATCH_CMD_DEFINE_PORT..	Defines a patch input or output port.	APR-147
PATCH_CMD_END	Marks the end of a PatchCmd list.	APR-148
PATCH_CMD_EXPOSE	Exposes a patch port.	APR-149
PATCH_CMD_JUMP	Links to another list of PatchCmds.	APR-150
PATCH_CMD_NOP	Skip this command.	APR-151
PATCH_CMD_SET_COHERENCE	Controls signal phase coherence along internal patch connections.	APR-152
PATCH_CMD_SET_CONSTANT	Sets a block input or knob to a constant.	APR-153

PatchCmdBuilder

AddTemplateToPatch	Adds a \4PATCH_CMD_ADD_TEMPLATE\4P PatchCmd to PatchCmdBuilder.	APR- 154
ConnectPatchPorts	Adds a \4PATCH_CMD_CONNECT\4P PatchCmd to PatchCmdBuilder.	APR-155
CreatePatchCmdBuilder	Creates a PatchCmdBuilder (convenience environment for constructing a \4PatchCmd\4P List)	APR-157
DefinePatchKnob	Adds a \4PATCH_CMD_DEFINE_KNOB\4P PatchCmd to PatchCmdBuilder.	APR-159
DefinePatchPort	Adds a \4PATCH_CMD_DEFINE_PORT\4P PatchCmd to PatchCmdBuilder.	APR-160
DeletePatchCmdBuilder	Deletes a PatchCmdBuilder created by \4CreatePatchCmdBuilder()\4P. ..	APR-161
ExposePatchPort	Adds a \4PATCH_CMD_EXPOSE\4P PatchCmd to PatchCmdBuilder.	APR-162
GetPatchCmdBuilderError	Returns error code from a failed PatchCmd constructor.	APR-163
GetPatchCmdList	Returns PatchCmd list from a PatchCmdBuilder.	APR-164
SetPatchCoherence	Adds a \4PATCH_CMD_SET_COHERENCE\4P PatchCmd to PatchCmdBuilder.	APR-165
SetPatchConstant	Adds a \4PATCH_CMD_SET_CONSTANT\4P PatchCmd to PatchCmdBuilder.	APR-166

4

Beep Folio Calls

ConfigureBeepChannel.....	Configure a DMA channel.	APR- 169
GetBeepTime	Return the current Beep time.	APR-170
LoadBeepMachine	Load the Beep Machine DSP program into DSP.	APR-171
SetBeepChannelData.....	Specify data for a DMA channel.	APR-172
SetBeepChannelDataNext.....	Specify data to play after current data finishes.	APR-173
SetBeepParameter	Set a global Beep parameter.	APR-175
SetBeepVoiceParameter	Set a Beep voice parameter.	APR-176
StartBeepChannel	Start a DMA channel.	APR-177
StopBeepChannel.....	Stop a DMA channel.	APR-178
UnloadBeepMachine	Unload the Beep Machine from the DSP.	APR-179

Machines

basic.bm	Basic Beep Machine.	APR-180
----------------	--------------------------	---------

5

DSP Instruments

--DSP-Instrument-Overview--	Overview of DSP Instrument documentation.	APR-185
Mixer	General description of custom mixer \f4Template\fpPs built by \f4CreateMixerTemplate()\fP.	APR-187

Accumulator

add_accum.dsp	Adds signed signal to DSP accumulator.	APR-189
input_accum.dsp	Loads the accumulator from an input.	APR-190
multiply_accum.dsp.....	Multiplies accumulator by a signed signal.	APR-191
output_accum.dsp	Stores accumulator to an output.	APR-192
subtract_accum.dsp	Subtracts accumulator from input.	APR-193
subtract_from_accum.dsp.....	Subtracts input from accumulator.	APR-194

Arithmetic

add.dsp	Adds two signed signals.	APR-195
expmod_unsigned.dsp	Exponential modulation of an unsigned signal.	APR-196
latch.dsp	Passes its input to output if gate held above zero.	APR-197
maximum.dsp	Picks the maximum of two input signals.	APR-198
minimum.dsp	Picks the minimum of two input signals.	APR-199
multiply.dsp	Multiplies two signed input signals (ring modulator).	APR-200
multiply_unsigned.dsp	Multiplies two unsigned signals.	APR-201
schmidt_trigger.dsp.....	Comparator with hysteresis and trigger.	APR-202

subtract.dsp.....	Returns the difference between two signed signals.....	APR-203
timesplus.dsp.....	Single-operation multiply and accumulate ($A*B+C$).....	APR-204
timesplus_noclip.dsp.....	Single-operation, unclipped multiply and accumulate ($A*B+C$).....	APR-205
times_256.dsp.....	Fast multiplication by 256.....	APR-206

Control_Signal

envelope.dsp.....	Interpolate a segment of an $\sqrt{4}$ Envelope\fp.....	APR-207
envfollower.dsp.....	Tracks the contour of a signal.....	APR-208
integrator.dsp.....	Integrates an input signal (ramp generator).....	APR-209
pulse_lfo.dsp.....	Pulse wave Low-Frequency Oscillator.....	APR-211
randomhold.dsp.....	Generates random values and holds them.....	APR-212
rednoise_lfo.dsp.....	Red noise LFO generator.....	APR-213
slew_rate_limiter.dsp.....	Slew rate limiter.....	APR-214
square_lfo.dsp.....	Square wave Low-Frequency Oscillator.....	APR-215
triangle_lfo.dsp.....	Triangle wave Low-Frequency Oscillator.....	APR-216

Diagnostic

benchmark.dsp.....	Outputs current DSPP tick count.....	APR-217
--------------------	--------------------------------------	---------

Effects

cubic_amplifier.dsp.....	Non-linear amplifier (distortion effect).....	APR-218
deemphcd.dsp.....	CD de-emphasis filter.....	APR-219
delay4.dsp.....	Delays an input signal by up to 4 frames.....	APR-220
delay_f1.dsp.....	Sends mono input to a delay line.....	APR-221
delay_f2.dsp.....	Sends stereo input to a delay line.....	APR-222
depopper.dsp.....	Helps eliminate pops when switching sounds.....	APR-223
filter_1o1z.dsp.....	First Order, One Zero filter.....	APR-224
filter_3d.dsp.....	Sound spatialization filter.....	APR-225
svfilter.dsp.....	State-variable digital filter.....	APR-226

Line_In_And_Out

line_in.dsp.....	Taps stereo audio line in.....	APR-228
line_out.dsp.....	Adds to stereo signal to send to audio line out.....	APR-229
tapoutput.dsp.....	Taps accumulated stereo line out signal.....	APR-230

Sampled_Sound

sampler_16_f1.dsp.....	Fixed-rate mono 16-bit sample player.....	APR-231
sampler_16_f2.dsp.....	Fixed-rate stereo 16-bit sample player.....	APR-232
sampler_16_v1.dsp.....	Variable-rate mono 16-bit sample player.....	APR-233
sampler_16_v2.dsp.....	Variable-rate stereo 16-bit sample player.....	APR-234

sampler_8_f1.dsp	Fixed-rate mono 8-bit sample player.....	APR-235
sampler_8_f2.dsp	Fixed-rate stereo 8-bit sample player.....	APR-236
sampler_8_v1.dsp	Variable-rate mono 8-bit sample player.....	APR-237
sampler_8_v2.dsp	Variable-rate stereo 8-bit sample player.....	APR-238
sampler_adp4_v1.dsp	Variable-rate mono sample player with ADPCM Intel/DVI 4:1 decompression.....	APR-239
sampler_cbd2_f1.dsp	Fixed-rate mono sample player with CBD2 2:1 decompression.....	APR-240
sampler_cbd2_f2.dsp	Fixed-rate stereo sample player with CBD2 2:1 decompression.....	APR-241
sampler_cbd2_v1.dsp	Variable-rate mono sample player with CBD2 2:1 decompression.....	APR-242
sampler_cbd2_v2.dsp	Variable-rate stereo sample player with CBD2 2:1 decompression.....	APR-243
sampler_drift_v1.dsp	Variable-rate mono 16-bit sample player with drift output (suitable for flanging).....	APR-244
sampler_raw_f1.dsp	Fixed-rate mono 16-bit sample player without amplitude scaling.....	APR-245
sampler_sqs2_f1.dsp	Fixed-rate mono sample player with SQS2 2:1 decompression.....	APR-246
sampler_sqs2_v1.dsp	Variable-rate mono sample player with SQS2 2:1 decompression.....	APR-247

Sound_Synthesis

chaos_1d.dsp	One-dimensional chaotic function generator (the logistic map).....	APR-248
impulse.dsp	Impulse waveform generator.....	APR-249
noise.dsp	White noise generator.....	APR-250
pulse.dsp	Pulse wave generator.....	APR-251
rednoise.dsp	Red noise generator.....	APR-252
sawtooth.dsp	Sawtooth wave generator.....	APR-253
square.dsp	Square wave generator.....	APR-254
triangle.dsp	Triangle wave generator.....	APR-255

6

Music Link Library Calls

ATAG

--ATAG-File-Format--	Simple audio object file format for Samples, Envelopes, Delay lines, and Tunings.....	APR-259
AudioTagHeaderSize	Returns size of AudioTagHeader for given number of tags.....	APR-261
LoadATAG.....	Load an ATAG simple audio object format file.....	APR-262
ValidateAudioTagHeader	Validate the contents of an AudioTagHeader (ATAG chunk).....	APR-263

Juggler_Classes

CollectionClass	Multiple parallel sequences and collections.....	APR-264
JuggleeClass.....	Root juggler class.....	APR-265
SequenceClass.....	A single sequence of events.....	APR-266

Juggler_Functions

AbortObject.....	Abnormally stops a juggler object.	APR-267
AllocObject.....	Allocates memory for an object.	APR-268
BumpJuggler.....	Bumps the juggler data-structure index.	APR-269
CreateObject.....	Creates an object of the given class.	APR-271
DefineClass.....	Defines a class of objects.	APR-272
DestroyObject.....	Destroys an object.	APR-273
FreeObject.....	Frees memory allocated for an object.	APR-274
GetNthFromObject.....	Gets the nth element of a collection.	APR-275
GetObjectInfo.....	Gets the current settings of an object.	APR-276
InitJuggler.....	Initializes the juggler mechanism for controlling events.	APR-277
PrintObject.....	Prints debugging information about an object.	APR-278
RemoveNthFromObject.....	Removes the nth element of a collection.	APR-279
SetObjectInfo.....	Sets values in the object based on tag args.	APR-280
StartObject.....	Starts an object so the juggler will play it.	APR-282
StopObject.....	Stops an object so the juggler won't play it.	APR-283
TermJuggler.....	Terminates the juggler mechanism for controlling events.	APR-284
ValidateObject.....	Validates an object.	APR-285

Sample

DeleteSampleInfo.....	Frees SampleInfo.	APR-286
DumpSampleInfo.....	Dumps SampleInfo structure returned by \f4GetAIFFSampleInfo()\fP.	APR-287
GetAIFFSampleInfo.....	Parses an AIFF sample file and return a SampleInfo data structure.	APR-288
LoadSample.....	Loads a \f4Sample\fP from an AIFF or AIFC file.	APR-290
LoadSystemSample.....	Loads a system \f4Sample\fP file.	APR-292
SampleFormatToInsName.....	Builds the name of the DSP Instrument Template to play a sample of the specified format.	APR-294
SampleInfoToTags.....	Fills out a tag list to create a \f4Sample\fP Item from SampleInfo.	APR-296
SampleItemToInsName.....	Builds the name of the DSP Instrument Template to play the \f4Sample\fP.	APR-298
UnloadSample.....	Unloads a sample loaded by \f4LoadSample()\fP.	APR-300

Score

ChangeScoreControl.....	Changes a MIDI control value for a channel.	APR-301
ChangeScorePitchBend.....	Changes a channel's pitch bend value.	APR-303
ChangeScoreProgram.....	Changes the MIDI program for a channel.	APR-305
ConvertPitchBend.....	Converts a MIDI pitch bend value into frequency multiplier.	APR-306
CreateScoreContext.....	Allocates a score context.	APR-308
CreateScoreMixer.....	Creates and initializes a mixer instrument for MIDI score playback.	APR-309

DeleteScoreContext.....	Deletes a score context.....	APR-311
DeleteScoreMixer.....	Disposes of mixer created by \f4CreateScoreMixer()\fP.....	APR-312
DisableScoreMessages.....	Enable or disable printed messages during score playback.....	APR-313
FreeChannelInstruments.....	Frees all of a MIDI channel's instruments.....	APR-314
GetScoreBendRange.....	Gets the current pitch bend range value for a score context.....	APR-315
InitScoreDynamics.....	Sets up dynamic voice allocation.....	APR-316
InterpretMIDIEvent.....	Interprets MIDI events within a MIDI object.....	APR-317
InterpretMIDIMessage.....	Executes a MIDI message.....	APR-318
LoadPIMap.....	Loads a Program-Instrument Map (\f4PIMap\fP) from a text file.....	APR-320
LoadScoreTemplate.....	Loads instrument or patch \f4Template\fP depending on file name extension.....	APR-321
MFLoadCollection.....	Creates a juggler collection from a MIDI file image in RAM.....	APR-323
MFLoadCollection.....	Loads a set of sequences from a MIDI file.....	APR-324
MFLoadSequence.....	Loads a sequence from a MIDI file.....	APR-325
MFLoadCollection.....	Unloads a MIDI collection.....	APR-326
NoteOffIns.....	Turns off a note played by an instrument.....	APR-327
NoteOnIns.....	Turns on a note for an instrument.....	APR-328
PIMap.....	Program-Instrument Map file format.....	APR-329
PurgeScoreInstrument.....	Purges an unused instrument from a ScoreContext.....	APR-331
ReleaseScoreNote.....	Releases a MIDI note, uses voice allocation.....	APR-333
SetPIMapEntry.....	Specifies the instrument to use when a MIDI program change occurs.....	APR-334
SetScoreBendRange.....	Sets the current pitch bend range value for a score context.....	APR-336
StartScoreNote.....	Starts a MIDI note, uses voice allocation.....	APR-337
StopScoreNote.....	Stop a MIDI note immediately with no release phase.....	APR-338
UnloadPIMap.....	Unloads instrument templates loaded previously with PIMap file.....	APR-339
UnloadScoreTemplate.....	Unloads a \f4Template\fP loaded by \f4LoadScoreTemplate()\fP.....	APR-340

Sound3D

Create3DSound.....	Allocate and initialize the 3D Sound data structures.....	APR-341
Delete3DSound.....	Release resources and deallocate 3D Sound data structures.....	APR-345
Get3DSoundInstrument.....	Returns the DSP instrument to which a source should be connected to use a 3D Sound.....	APR-346
Get3DSoundParms.....	Return application-dependent cue parameters.....	APR-347
Get3DSoundPos.....	Calculate the instantaneous position of a 3D Sound at the time of the call.....	APR-349
Move3DSound.....	Moves a 3D Sound in a line from start to end over a given time period....	APR-350
Move3DSoundTo.....	Moves a 3D Sound in a line from wherever it currently is to a given endpoint.....	APR-352

s3dNormalizeAngle	Normalize an angle to be between -PI and +PI	APR-353
Start3DSound	Starts a 3D Sound, and positions it in space.	APR-354
Stop3DSound	Stops a 3D Sound.	APR-355

SoundPlayer

spAddMarker	Add a new SPMarker to an SPSound.	APR-356
spAddSample	Create an SPSound for a \f4Sample\fp Item.	APR-357
spAddSoundFile	Create an SPSound for an AIFF sound file.	APR-360
spBranchAtMarker	Set up a static branch at a marker.	APR-363
spClearDefaultDecisionFunction	Clears global decision function.	APR-365
spClearMarkerDecisionFunction	Clears a marker decision function.	APR-366
spContinueAtMarker	Clear static branch at a marker.	APR-367
spCreatePlayer	Create an SPPlayer.	APR-368
SPDecisionFunction	Typedef for decision callback functions.	APR-371
spDeletePlayer	Delete an SPPlayer.	APR-374
spDumpPlayer	Print debug information for sound player.	APR-375
spFindMarkerName	Return pointer an SPMarker by looking up its name.	APR-376
spGetMarkerName	Get name of an SPMarker.	APR-377
spGetMarkerPosition	Get position of an SPMarker.	APR-378
spGetPlayerFromSound	Get SPPlayer that owns an SPSound.	APR-379
spGetPlayerSignalMask	Get set of signals player will send to client.	APR-380
spGetPlayerStatus	Get current status of an SPPlayer.	APR-381
spGetSoundFromMarker	Get SPSound that owns an SPMarker.	APR-382
spIsSoundInUse	Determines if SPPlayer is currently reading from a particular SPSound.	APR-383
spLinkSounds	Branch at end of one sound to beginning of another.	APR-384
spLoopSound	Branch at end of sound back to the beginning.	APR-385
spPause	Pause an SPPlayer.	APR-386
spRemoveMarker	Manually remove an SPMarker from an SPSound.	APR-387
spRemoveSound	Manually remove an SPSound from an SPPlayer.	APR-388
spResume	Resume playback of an SPPlayer after being paused.	APR-390
spService	Service SPPlayer.	APR-391
spSetBranchAction	Set up an SPAction to branch to the specified marker.	APR-393
spSetDefaultDecisionFunction	Install a global decision function to be called for every marker.	APR-394
spSetMarkerDecisionFunction	Install a marker decision function.	APR-395
spSetStopAction	Set up an SPAction to stop reading.	APR-396
spStartPlaying	Begin emitting sound for an SPPlayer.	APR-397
spStartReading	Start SPPlayer reading from an SPSound.	APR-399
spStop	Stops an SPPlayer.	APR-401
spStopAtMarker	Stop when playback reaches marker.	APR-402

SoundSpooler

SoundBufferFunc	SoundSpooler callback function typedef	APR-403
ssplAbort	Aborts SoundSpooler buffers in active queue	APR-405
ssplAttachInstrument	Attaches new sample player instrument to SoundSpooler	APR-406
ssplCreateSoundSpooler	Creates a SoundSpooler	APR-407
ssplDeleteSoundSpooler	Deletes a SoundSpooler	APR-408
ssplDetachInstrument	Detaches the current sample player instrument from the SoundSpooler ..	APR-409
ssplDumpSoundBufferNode	Print debug info for SoundBufferNode	APR-410
ssplDumpSoundSpooler	Print debug info for SoundSpooler	APR-411
ssplGetSBMsgClass	Get class of message passed to <code>\f4SoundBufferFunc\fp</code>	APR-412
ssplGetSequenceNum	Gets Sequence Number of a SoundBufferNode	APR-413
ssplGetSpoolerStatus	Gets SoundSpooler status flags	APR-414
ssplGetUserData	Gets User Data from a SoundBufferNode	APR-415
ssplIsSpoolerActive	Tests if spooler has anything in its active queue	APR-416
ssplPause	Pauses the SoundSpooler	APR-417
ssplPlayData	Waits for next available buffer then sends a block full of data to the spooler (convenience function)	APR-418
ssplProcessSignals	Processes completion signals that have been received	APR-419
ssplRequestBuffer	Asks for an available buffer	APR-421
ssplReset	Resets SoundSpooler	APR-423
ssplResume	Resume SoundSpooler playback	APR-424
ssplSendBuffer	Sends a buffer full of data	APR-425
ssplSetBufferAddressLength	Attaches sample data to a SoundBufferNode	APR-426
ssplSetSoundBufferFunc	Install new <code>\f4SoundBufferFunc\fp</code> in SoundSpooler	APR-427
ssplSetUserData	Stores user data in a SoundBufferNode	APR-428
ssplSpoolData	Sends a block full of data. (convenience function)	APR-429
ssplStartSpoolerTags	Starts SoundSpooler	APR-430
ssplStopSpooler	Stops SoundSpooler	APR-431
ssplUnrequestBuffer	Returns an unused buffer to the sound spooler	APR-432
UserBufferProcessor	Callback function prototype called by <code>\f4ssplProcessSignals()\fp</code> , <code>\f4ssplAbort()\fp</code> , and <code>\f4ssplReset()\fp</code>	APR-433

Preface

About This Book

3DO M2 Audio Programmer's Reference describes the audio and music calls available for programmers of the M2 system. These calls are listed in alphabetical order in each chapter.

About the Audience

This book is written for programmers and developers writing applications for the 3DO M2 system. To use this document, you should have a working knowledge of the C programming language, object-oriented concepts, and, of course, music and audio.

How This Book Is Organized

This book includes the following chapters:

Chapter 1, Audio Folio Calls — Lists the Audio Folio function calls.

Chapter 2, Audio Patch File Folio Calls — Lists the Audio Patch File Folio function calls and describes the binary patch file format.

Chapter 3, Audio Patch Folio Calls — Lists the Audio Patch Folio calls and patch command set.

Chapter 4, Beep Folio Calls — Lists the Beep Folio calls and Beep Machines.

Chapter 5, DSP Instrument Calls — Lists the DSP instrument templates available for the Audio Folio.

Chapter 6, Music Link Library Calls — Lists the music link library calls.

Related Documentation

The following manuals are useful to developers who are programming in the 3DO environment:

3DO M2 System Programmer's Guide. A guide to the features of the kernel, IO, filesystem, and so on in the 3DO operating system.

3DO M2 System Programmer's Reference. Contains a detailed description of the calls that make up the Portfolio system.

3DO M2 Music and Audio Programmer's Guide Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Debugger Programmer's Guide. A guide to using the 3DO Debugger. It provides a tutorial on using the debugger as well as descriptions of the graphical user interface.

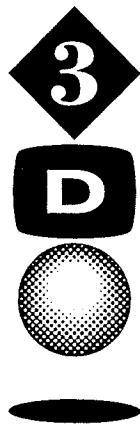
3DO M2 DataStreamer Programmer's Guide. A guide to the 3DO DataStreamer architecture, streaming tools, and weaver tool.

3DO M2 DataStreamer Programmer's Reference. Provides manual pages for the structures in the 3DO DataStreamer library, for all data preparation tools, and for all weaver script commands.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>int32 OpenGraphicsFolio(void)</code>
procedure name	<code>CreateScreenGroup()</code>
new term or emphasis	A <i>ViewList</i> is a special case of a View.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.



3DO M2 Audio Programmer's Reference

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Chapter 1

Audio Folio Calls

This section presents the reference documentation for the Audio folio and associated link libraries.

--Audio-Port-Types--

Audio port type descriptions.

Description

All Instrument ports have one of the port types defined below. The port type of any port may be read with `GetInstrumentPortInfoByName()` or `GetInstrumentPortInfoByIndex()`.

Port Types

AF_PORT_TYPE_INPUT

These ports may be the destination of a connection created with `ConnectInstrumentParts()` and `PATCH_CMD_CONNECT`. They may be created within a patch by `PATCH_CMD_DEFINE_PORT`. These ports may be single- or multi-part and may have any signal type.

AF_PORT_TYPE_OUTPUT

These ports may be the source of a connection created with `ConnectInstrumentParts()` and `PATCH_CMD_CONNECT`, and may also be examined with a Probe. They may be created within a patch by `PATCH_CMD_DEFINE_PORT`. These ports may be single- or multi-part and may have any signal type.

AF_PORT_TYPE_KNOB

Knobs may be set manually (via `CreateKnob()` and `SetKnobPart()`) or be the destination of a connection created with `ConnectInstrumentParts()` and `PATCH_CMD_CONNECT`. They may be created within a patch by `PATCH_CMD_DEFINE_KNOB`. Knobs may be single- or multi-part and may have any signal type.

AF_PORT_TYPE_IN_FIFO

These ports are for attaching a Samples to be played. They are always single-part and have no signal type.

AF_PORT_TYPE_OUT_FIFO

These ports are for attaching a delay line Sample to be written to by a delay instrument (e.g., `delay_f1.dsp`). They are always single-part and have no signal type.

AF_PORT_TYPE_TRIGGER

Triggers are the means of signaling the host CPU by the DSP. They are always single-part and have no signal type. See `schmidt_trigger.dsp` and `ArmTrigger()`.

AF_PORT_TYPE_ENVELOPE

These ports are for attaching Envelopes. They are always single-part. Envelope hooks have no signal type, but Envelope items do. Envelope hooks are always accompanied by a pair of knobs (with the same name as the envelope hook plus the suffixes `.request` and `.incr`) which may be controlled manually instead of with an Envelope. See `envelope.dsp` for details.

Associated Files

<:audio:audio.h>

See Also

--Audio-Signal-Types--

--Audio-Signal-Types--

Audio signal type descriptions.

Description

Instrument ports (Knobs, Inputs, and Outputs), Knobs, Probes, and Envelopes all have a signal type, which defines the units and legal ranges for their values. The signal type of an instrument port may be read with `GetInstrumentPortInfoByName()` or `GetInstrumentPortInfoByIndex()`. Knobs and Probes inherit the signal type of the port they are attached to by default, but may be set to any other signal when created. An Envelope's signal type determines the units of the `envs_Value` members of its `EnvelopeSegment` array.

Since the DSP only operates on generic signals, the others are merely convenient representations. The conversion formula from generic to each non-generic type is listed under the description for that type. In each formula, `Generic` represents the internal generic signal, `SystemSampleRate` represents the DAC sample rate, and `CalcRateDivision` is the `AF_TAG_CALCRATE_DIVIDE` value for the instrument (either 1, 2, or 8). The functions `ConvertGenericToAudioSignal()` and `ConvertAudioSignalToGeneric()` perform conversions between generic and non-generic signal types.

Signal Type Descriptions**AUDIO_SIGNAL_TYPE_GENERIC_SIGNED**

Signed generic signal (e.g., amplitude).

AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED

Unsigned generic signal.

AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz.

$$\text{Freq (Hz)} = \text{Generic} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0)$$

AUDIO_SIGNAL_TYPE_LFO_FREQ

Low-frequency oscillator frequency in Hertz.

$$\text{Freq (Hz)} = \text{Generic} * \text{SystemSampleRate} / (\text{CalcRateDivision} * 2.0 * 256)$$

AUDIO_SIGNAL_TYPE_SAMPLE_RATE

Sample rate in samples/second.

$$\text{SampleRate (Hz)} = \text{Generic} * \text{SystemSampleRate} / \text{CalcRateDivision}$$

AUDIO_SIGNAL_TYPE_WHOLE_NUMBER

Whole numbers in the range of -32768 to 32767.

$$\text{Whole} = \text{Generic} * 32768.0$$

Signal Ranges And Precision

Signal Type	Min	Max
Precision Units		
-----	-----	-----
AUDIO_SIGNAL_TYPE_GENERIC_SIGNED	-1.00000	0.99997
3.05176e-05		
AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED	0.00000	1.99997
3.05176e-05		

AUDIO_SIGNAL_TYPE_OSC_FREQ	-22050.00000	22049.32617
0.67291 Hz *		
AUDIO_SIGNAL_TYPE_LFO_FREQ	-86.13281	86.13018
2.62856e-03 Hz *		
AUDIO_SIGNAL_TYPE_SAMPLE_RATE	0.00000	88198.65625
1.34583 samp/s *		
AUDIO_SIGNAL_TYPE_WHOLE_NUMBER	-32768.00000	32767.00000
1.00000		

* Actual frequency ranges and precision are determined by instrument execution

rate, which is determined by DAC sample rate (normally 44100) and instrument

calculation rate division. As presented, the instrument execution rate is

assumed to be 44100 times/second.

Associated Files

<:audio:audio_signals.h>, <:audio:audio.h>

See Also

--Audio-Port-Types--, ConvertAudioSignalToGeneric(),
ConvertGenericToAudioSignal(), GetAudioSignalInfo()

CreateAttachment

Creates and Attachment between a Sample or Envelope to an Instrument or Template.

Synopsis

```
Item CreateAttachment (Item master, Item slave, const TagArg  
*tagList)
```

```
Item CreateAttachmentVA (Item master, Item slave, uint32 tag1, ...)
```

Description

This procedure connects a Sample or Envelope (slave Item) to an Instrument or Template (master Item). This sample or envelope will be played when the instrument is started. If the procedure is successful, it returns an attachment item number.

When you finish with the attachment, you should call `DeleteAttachment()` to detach and free the attachment's resources. Deleting either the master or slave Item does this automatically.

Some Instruments may have multiple places where Envelopes or Samples can be attached. In that case you must specify which attachment point (the hook) is to be used. An instrument may, for example, have an amplitude envelope and a filter envelope, which may be called `AmpEnv` and `FilterEnv` respectively. You can specify which hook to use by passing its name with the `AF_TAG_NAME` tag.

If you do not specify a hook name, then the name "Env" is assumed for envelopes. For samples, it is assumed that you wish use the only sample hook available. Sample players have a hook called "InFIFO". Delay instruments have a hook called "OutFIFO". If you do not specify a hook name for a sample, and the instrument has more than one sample hook then `AF_ERR_BAD_NAME` is returned.

You may use the same envelope for several instruments. The envelope data may be edited at any time, but be aware that it is read by a high priority task in the audio folio. Thus you should not leave it in a potentially "goofy" state if it is actively used.

If multiple Samples are attached to a FIFO, and the instrument is started with a specified pitch, then the list of samples is searched for the first sample whose range of notes and velocities matches the desired note index (pitch). The `LowNote` and `HighNote`, and `LowVelocity` and `HighVelocity` are read from the AIFF file. The values from the file can be overwritten using `SetAudioItemInfo()` on each Sample.

By default when an Attachment is deleted, the slave Item is unaffected. An Attachment created with { `AF_TAG_AUTO_DELETE_SLAVE`, `TRUE` } causes its slave to be automatically deleted when the Attachment is deleted. Because Attachments themselves are automatically deleted when either the master or slave is deleted, the auto-deletion effect can be triggered by deleting either the master or slave of such an attachment.

Attachments made to a Template are automatically propagated to Instruments created from that Template. All properties are copied from Template Attachments to Instrument Attachments except for the `AF_TAG_AUTO_DELETE_SLAVE`, which is set to `FALSE`. `AF_TAG_AUTO_DELETE_SLAVE` isn't propagated because the propagation duplicates just the Template's Attachment items, not the slave items. This permits deleting instruments created from such a template without deleting the slave items attached to the template.

Arguments

master

The item number of the Instrument or Template.

slave

The item number of the Sample or Envelope.

Tag Arguments

See Attachment for legal tags.

Return Value

The procedure returns the item number of the Attachment (a non-negative value) or an error code a negative value) if an error occurs.

Implementation

Convenience call implemented in libc.a V29.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_ATTACHMENT_NODE) ,  
              AF_TAG_MASTER, master,  
              AF_TAG_SLAVE, slave,  
              TAG_JUMP, tagList);
```

Associated Files

<:audio:audio.h>, libc.a

See Also

DeleteAttachment(), Attachment, Instrument, Template, Envelope, Sample

Caveats

!!! The hook name of an Attachment made to a Template isn't checked until an instrument is allocated from the Template. CreateAttachment() to a template with an invalid hook name will succeed, but CreateInstrument() from that template will fail. CreateInstrument() from a template in this condition has been known to leave the system in an unstable, crash-prone state.

DeleteAttachment

Undoes Attachment between Sample or Envelope and Instrument or Template.

Synopsis

```
Err DeleteAttachment (Item attachment)
```

Description

This procedure disconnects a Sample or an Envelope from an Instrument or Template, deletes the attachment item connecting the two, and frees the attachment's resources. If the attachment was created with { AF_TAG_AUTO_DELETE_SLAVE, TRUE }, then the slave Sample or Envelope is also deleted. Otherwise, the slave item is unaffected.

Arguments

attachment
The item number of the Attachment.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:audio.h> V27.

Associated Files

<:audio:audio.h>

See Also

CreateAttachment(), Attachment

GetAttachments

Returns list of Attachments to a sample or envelope hook.

Synopsis

```
int32 GetAttachments (Item *attachmentsr, int32 maxAttachments,  
                    Item insOrTemplate, const char *hookName)
```

Description

This function returns an array of Attachment Items made to a sample or envelope hook of an Instrument or instrument Template.

Arguments

attachments

A pointer to an array of Items where the attachment list will be stored. Can be NULL if maxAttachments is 0.

maxAttachments

Maximum number of attachments to write to the attachments array. Can be 0, in which case no attachments will be written to the attachments array.

insOrTemplate

Instrument or Instrument Template Item to query.

hookName

Envelope or Sample hook of instrument to query.

Return Value

≥ 0

Number of attachments to hook. The number of elements written to attachments is $\text{MIN}(\text{return value}, \text{maxAttachments})$.

< 0

Error code. attachments array state is undefined.

Implementation

Folio call implemented in audio folio V30.

Associated Files

`<:audio:audio.h>`

See Also

`GetNumAttachments()`, `GetInstrumentPortInfoByName()`, `insinfo`

GetNumAttachments

Returns the number of Attachments to a sample or envelope hook.

Synopsis

```
int32 GetNumAttachments (Item insOrTemplate, const char *hookName)
```

Description

This function returns the number of Attachment Items made to a sample or envelope hook of an Instrument or instrument Template.

Arguments

insOrTemplate

Instrument or Instrument Template Item to query.

hookName

Envelope or Sample hook of instrument to query.

Return Value

Non-negative value indicating the number of attachments to the specified hook, or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>` V30.

Notes

This macro is equivalent to:

```
GetAttachments (NULL, 0, insOrTemplate, hookName);
```

Associated Files

`<:audio:audio.h>`

See Also

```
GetAttachments(), GetInstrumentPortInfoByName()
```


LinkAttachments

Connects Sample Attachments for sequential playback.

Synopsis

```
Err LinkAttachments (Item Att1, Item Att2)
```

Description

This procedure specifies that the attachment Att2 will begin playing when attachment Att1 finishes. This is useful if you want to connect discontinuous sample buffers that are used for playing interleaved audio and video data. It is also a good way to construct big sound effects from a series of small sound effects.

If Att1's sample has a sustain loop, but no release loop, you can follow this with a call to `ReleaseAttachment(Att1,NULL)` to smoothly transition to Att2.

If Att1's sample has no loops, Att2 will automatically start as soon as Att1 completes (assuming that it has not completed prior to this function being called).

If, after linking Att1 to Att2, `StopAttachment()` is called on Att1 before Att1 finishes, Att2 will not be automatically started. `StopAttachment()` on Att1 after Att1 finishes has no effect.

All links remain in effect for multiple calls to `StartInstrument()` or `StartAttachment()`. That is, if you call `LinkAttachments(Att1,Att2)`, Att1 will flow into Att2 upon completion of Att1 for every subsequent call to `StartAttachment(Att1,NULL)` (or `StartInstrument()` on the instrument belonging to Att1) if Att1 would normally be automatically started by starting the instrument).

An attachment (Att1) can link to no more than one attachment. An attachment (Att2) can be linked to multiple attachments. The most recent call to `LinkAttachments()` for Att1 takes precedence.

The pair of Attachments passed to this function must satisfy all of these requirements:

- * Both Attachments must be Sample Attachments.
- * Both Attachments must be attached to the same Instrument.
- * Both Attachments must be attached to the same FIFO of that Instrument.

Call `LinkAttachments(Att1,0)` to remove a previous link from Att1.

A link does not interfere with a Cue associated with an attachment.

Deleting either Attachment, either Attachment's Sample, or the Instrument to which the Attachment belongs, breaks the link.

Arguments

Att1

The item number for the attachment that is to finish. Must be a Sample Attachment.

Att2

The item number for the attachment that is to begin playing. Must be a Sample Attachment attached to the same FIFO on the same Instrument as Att1. Can be 0 to remove a link from Att1.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Caveats

In order for Att2 to start, Att1 must either be currently playing or yet to be played. If it has already completed by the time of this call, Att2 will not be started.

Associated Files

<:audio:audio.h>, libc.a

See Also

StartAttachment(), ReleaseAttachment()

MonitorAttachment

Monitors an Attachment, sends a Cue at a specified point.

Synopsis

```
Err MonitorAttachment (Item Attachment, Item Cue, int32 Index)
```

Description

This procedure sends a Cue to the calling task when the specified Attachment reaches the specified point. The procedure is often used with a sample Attachment to send a cue when the sample has been fully played.

There can be only one Cue per Attachment. The most-recent call to `MonitorAttachment()` takes precedence.

To remove the current Cue from an Attachment, call `MonitorAttachment(Attachment,0,0)`.

Arguments

Attachment

The item number for the Attachment.

Cue

Item number for the Cue to be associated with this attachment. Can be 0 to remove a the current Cue from the Attachment.

Index

Value indicating the point to be monitored. At this time, the only value that can be passed is `CUE_AT_END`, which asks that a Cue be sent at the end of an Attachment.

Index is ignored if Cue is 0.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

`GetCueSignal()`, `StartAttachment()`, `LinkAttachments()`, `WhereAttachment()`

ReleaseAttachment

Releases an Attachment.

Synopsis

```
Err ReleaseAttachment (Item Attachment, TagArg *tagList)
```

```
Err ReleaseAttachmentVA (Item Attachment, uint32 tag1, ...)
```

Description

This procedure releases an attachment and is commonly used to release attachments started with `StartAttachment()`. `ReleaseAttachment()` causes an attachment in a sustain loop to enter release phase. Has no effect on attachment with no sustain loop, or not in its sustain loop.

Arguments

Attachment
The item number for the attachment.

Tag Arguments

None

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Caveats

Prior to V24, `ReleaseAttachment()` did not support envelope attachments.

Associated Files

<:audio:audio.h>, libc.a

See Also

`StartAttachment()`, `StopAttachment()`, `LinkAttachments()`,
`ReleaseInstrument()`

StartAttachment

Starts an Attachment.

Synopsis

```
Err StartAttachment (Item Attachment, TagArg *tagList)
```

```
Err StartAttachmentVA (Item Attachment, uint32 tag1, ...)
```

Description

This procedure starts playback of an attachment, which may be an attached envelope or sample. This function is useful to start attachments that aren't started by `StartInstrument()` (e.g. attachments with the `AF_ATT_NOAUTOSTART` flag set).

An attachment started with `StartAttachment()` should be released with `ReleaseAttachment()` and stopped with `StopAttachment()` if necessary.

Arguments

Attachment

The item number for the attachment.

Tag Arguments

None

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

Caveats

Prior to V24, `StartAttachment()` did not support envelope attachments.

See Also

`ReleaseAttachment()`, `StopAttachment()`, `LinkAttachments()`,
`CreateAttachment()`, `StartInstrument()`

StopAttachment

Stops an Attachment.

Synopsis

```
Err StopAttachment (Item Attachment, TagArg *tagList)
```

```
Err StopAttachmentVA (Item Attachment, uint32 tag1, ...)
```

Description

This procedure abruptly stops an attachment. The attachment doesn't go into its release phase when stopped this way.

Arguments

Attachment
The item number for the attachment.

Tag Arguments

None

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Caveats

Prior to V24, StopAttachment () did not support envelope attachments.

Associated Files

<:audio:audio.h>, libc.a

See Also

StartAttachment (), ReleaseAttachment (), LinkAttachments (),
StopInstrument ()

WhereAttachment

Returns the current playing location of an Attachment.

Synopsis

```
int32 WhereAttachment (Item Attachment)
```

Description

This procedure is useful for monitoring the progress of a sample or envelope that's being played. It returns a value indicating where playback is located in the attachment's sample or envelope. For sample attachments, returns the currently playing byte offset within the sample. For envelope attachments, returns the currently playing segment index.

A sample's offset starts at zero. Note that the offset is not measured in sample frames. You must divide the byte offset by the number of bytes per frame, then discard the remainder to find out which frame is being played.

Arguments

Attachment

Item number of the attachment.

Return Value

Non-negative value indicating position (byte offset of sample or segment index of envelope) on success, negative error code on failure.

Implementation

Folio call implemented in audio folio V20.

Caveats

If a sample attachment has finished, the return value may be negative, or greater than the length of the sample. This is because the DMA hardware is pointing to a different sample. To determine whether an Attachment has finished, use `MonitorAttachment()`.

Associated Files

<:audio:audio.h>, libc.a

See Also

`MonitorAttachment()`

CreateCue

Creates an audio Cue.

Synopsis

```
Item CreateCue (const TagArg *tagList)
```

```
Item CreateCueVA (uint32 tag1, ...)
```

Description

Create an audio cue, which is an item associated with a system signal. A task can get the signal mask of the cue's signal by calling `GetCueSignal()`.

When a task uses an audio timing call such as `SignalAtAudioTime()`, it passes the item number of the audio cue to the procedure. The task then calls `WaitSignal()` to enter wait state, where it waits for the cue's signal at the specified time.

Since each task has its own set signals, Cues cannot be shared among tasks.

Call `DeleteCue()` to dispose of a Cue item.

Tag Arguments

\None

Return Value

The procedure returns the item number of the cue (a positive value) or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in the `<:audio:audio.h>` V21.

Notes

This function is equivalent to:

```
CreateItem (MKNODEID(AUDIONODE,AUDIO_CUE_NODE), tagList)
```

Associated Files

`<:audio:audio.h>`

See Also

`DeleteCue()`, `GetCueSignal()`, `MonitorAttachment()`, `SignalAtAudioTime()`, `ArmTrigger()`, `Cue`

DeleteCue

Deletes an audio Cue

Synopsis

```
Err DeleteCue (Item Cue)
```

Description

This macro deletes an audio cue and frees its resources (including its associated signal).

Arguments

Cue

The item number of the cue to delete.

Return Value

The procedure returns zero if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>` V21.

Associated Files

`<:audio:audio.h>`

See Also

`CreateCue()`

GetCueSignal

Returns a signal mask of a Cue.

Synopsis

```
int32 GetCueSignal (Item cue)
```

Description

This procedure returns a signal mask containing the signal bit allocated for an audio Cue item. The mask can be passed to `WaitSignal()` so a task can enter wait state until a Cue completes.

Arguments

cue
Item number of Cue.

Return Value

The procedure returns a signal mask (a positive value) if successful or a non-positive value (0 or an error code) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Caveats

This function has always returned 0 when a bad item is passed in. It should probably have returned `AF_ERR_BADITEM`, but will continue to return 0 for this case.

Associated Files

<:audio:audio.h>

See Also

`CreateCue()`

CreateEnvelope

Creates an Envelope.

Synopsis

```
Item CreateEnvelope (const EnvelopeSegment *points, int32 numPoints,  
                    const TagArg *tagList);
```

```
Item CreateEnvelopeVA (const EnvelopeSegment *points, int32  
numPoints,  
                     uint32 tag1, ...);
```

Description

Creates an Envelope Item which can be attached to an Instrument or Template with an envelope hook.

When you are finished with the Envelope, delete it with `DeleteEnvelope()`.

Arguments

points

An array of `EnvelopeSegment` values giving time in seconds accompanied by data values. The Points array is not copied: it must remain valid for the life of the Envelope Item or until it is replaced with another array of Points by a call to `SetAudioItemInfo()`.

numPoints

The number of points in the array.

Tag Arguments

!!! add description. see Envelope for now.

Return Value

The procedure returns the item number of the envelope (a non-negative value) or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in `libc.a V29`.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_ENVELOPE_NODE),  
             AF_TAG_ADDRESS, points,  
             AF_TAG_FRAMES,  numPoints,  
             TAG_JUMP,      tagList);
```

Associated Files

<:audio:audio.h>, `libc.a`

See Also

`Envelope`, `DeleteEnvelope()`, `CreateAttachment()`, `envelope.dsp`

DeleteEnvelope

Deletes an Envelope.

Synopsis

```
Err DeleteEnvelope (Item Envelope)
```

Description

This procedure deletes the specified envelope, freeing its resources. It also deletes any Attachments to the envelope.

If the Envelope was created with { AF_TAG_AUTO_FREE_DATA, TRUE }, then the EnvelopeSegment array is also be freed when the Envelope Item is deleted. Otherwise, the EnvelopeSegment array is not freed automatically.

Arguments

Envelope
Item number of the Envelope to delete.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro defined in `<:audio:audio.h>` V29.

Associated Files

`<:audio:audio.h>`

See Also

Envelope, CreateEnvelope(), DeleteAttachment()

AbandonInstrument

Makes an Instrument available for adoption from Template

Synopsis

```
Err AbandonInstrument (Item Instrument)
```

Description

This function together with `AdoptInstrument()` form a simple, but efficient, voice allocation system for a single instrument Template. `AbandonInstrument()` adds an instrument to a pool of unused instruments; `AdoptInstrument()` allocates instruments from that pool. A Template's pool can grow or shrink dynamically.

Why should you use this system when `CreateInstrument()` and `DeleteInstrument()` also do dynamic voice allocation? The answer is that `CreateInstrument()` and `DeleteInstrument()` create and delete Items and allocate and free DSP resources. `AdoptInstrument()` and `AbandonInstrument()` merely manage a pool of already existing Instrument Items belonging to a template. Therefore they don't have the overhead of Item creation and DSP resource management.

This function stops the instrument, if it was running, and sets its status to `AF_ABANDONED` (see `GetAudioItemInfo()` `AF_TAG_STATUS`). This instrument is now available to be adopted from its Template by calling `AdoptInstrument()`.

Instruments created with the `AF_INSF_AUTOABANDON` flag set, are automatically put into the `AF_ABANDONED` state when stopped.

Arguments

Instrument

The item number for the instrument to abandon.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

`<:audio:audio.h>`

See Also

`AdoptInstrument()`, `StopInstrument()`, `GetAudioItemInfo()`, `Instrument`

AdoptInstrument

Adopts an abandoned Instrument from a Template.

Synopsis

```
Item AdoptInstrument (Item instrumentTemplate)
```

Description

This function adopts an instrument from this template's abandoned instrument pool (finds an instrument belonging to the the template whose status is AF_ABANDONED). It then sets the instrument state to AF_STOPPED and returns the instrument item number. If the template has no abandoned instruments, this function returns 0.

Note that this function does not create a new Instrument Item; it returns an Instrument Item that was previously passed to AbandonInstrument() (or became abandoned because AF_INSF_AUTOABANDON was set).

This function together with AbandonInstrument() form a simple, but efficient, voice allocation system for a single instrument Template.

Arguments

instrumentTemplate

The item number for the instrument Template from which to attempt to adopt an Instrument.

Return Value

> 0

Instrument Item number if an instrument could be adopted.

0

If no abandoned instruments in template.

< 0

Error code on failure.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>

See Also

AbandonInstrument(), GetAudioItemInfo(), Instrument

BendInstrumentPitch

Bends an Instrument's output pitch up or down by a specified amount.

Synopsis

```
Err BendInstrumentPitch (Item instrument, float32 bendFrac)
```

Description

This procedure sets a new pitch bend value for an Instrument, a value that is stored as part of the Instrument. This function sets the resulting pitch to the product of the Instrument's frequency and BendFrac.

This setting remains in effect until changed by another call to `BendInstrumentPitch()`. To restore the original pitch of an Instrument, pass the result of 1.0 as `bendFrac`.

Use `Convert12TET_FP()` to compute a `bendFrac` value from an interval of semitones and cents.

This procedure won't bend pitch above the upper frequency limit of the instrument.

Arguments

`instrument`

The item number of an instrument

`bendFrac`

Factor to multiply Instrument's normal frequency by. This is a signed value; negative values result in a negative frequency value, which is not supported by all instruments. Specify 1.0 as `bendFrac` to restore the Instrument's original pitch.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

`<:audio:audio.h>`, `libc.a`

See Also

`Convert12TET_FP()`, `ConvertPitchBend()`

ConnectInstrumentParts

Patches the output of an Instrument to the input of another Instrument.

Synopsis

```
Err ConnectInstrumentParts (  
    Item srcInstrument, const char *srcPortName, int32 srcPartNum,  
    Item dstInstrument, const char *dstPortName, int32 dstPartNum)
```

Description

This procedure connects an output from one instrument to an input of another instrument. This allows construction of complex "patches" from existing synthesis modules. Some inputs and outputs have multiple parts, or channels, such as mixers, or stereo sample players. You can connect to a specific part by passing a part index. For connections that don't require a part index, pass zero, or use `ConnectInstruments()`

An output can be connected to one or more inputs; only one output can be connected to any given input. If you connect an output to a knob, it disconnects that knob from any possible control by `SetKnobPart()`. Unlike Attachments, this kind of connection does not create an Item.

You may call `DisconnectInstrumentParts()` to break a connection set up by this function. Connections are tracked by the audio folio, so it is not necessary to disconnect before deleting instruments.

See the DSP Instrument Templates chapter for complete listings of each template's ports.

Arguments

`srcInstrument`

Item number of the source instrument.

`srcPortName`

Name of the output port of the source instrument to connect to.

`srcPartNum`

Part index of output port of the source instrument to connect to. For ports with a single part, use zero.

`dstInstrument`

Item number of the destination instrument.

`dstPortName`

Name of the input port of the destination instrument to connect to.

`dstPartNum`

Part index of output port of the destination instrument to connect to. For ports with a single part, use zero.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Some of the possible error codes are:

AF_ERR_INUSE

If an instrument is already connected to the destination instrument's named input. You must disconnect the old instrument before connecting another.

AF_ERR_OUTOFRANGE

If either of the part numbers is out of range.

Implementation

Folio call implemented in audio folio V27.

Notes

Port names are matched case-insensitively.

Caveats

Be aware of the interactions between knob settings (even default knob settings) and connections. A connection takes precedence over any knob setting. When the knob is disconnected, any previous knob setting is restored. Because of connection tracking, deleting the source instrument also causes this sort of a restoring of knob settings.

Associated Files

<:audio:audio.h>

See Also

`DisconnectInstrumentParts()`, `ConnectInstruments()`

ConnectInstruments

Patches the output of an Instrument to the input of another Instrument.

Synopsis

```
Err ConnectInstruments (Item srcInstrument, const char *srcPortName,  
                        Item dstInstrument, const char *dstPortName)
```

Description

This is a convenience macro that can be used to connect single-part ports to one another.

Arguments

srcInstrument

Item number of the source instrument.

srcPortName

Name of the output port of the source instrument to connect to. Always connects to part 0.

dstInstrument

Item number of the destination instrument.

dstPortName

Name of the input port of the destination instrument to connect to. Always connects to part 0.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

AF_ERR_INUSE is returned if an instrument is already connected to the destination instrument's named input. You must disconnect the old instrument before connecting another.

Implementation

Macro call implemented in `<:audio:audio.h>` v27.

Notes

This macro is equivalent to:

```
ConnectInstrumentParts (srcInstrument, srcPortName, 0,  
                        dstInstrument, dstPortName, 0);
```

Associated Files

`<:audio:audio.h>`

See Also

`ConnectInstrumentParts()`

CreateInstrument

Allocates an Instrument from an Instrument Template.

Synopsis

```
Item CreateInstrument (Item insTemplate, const TagArg *tagList)
```

```
Item CreateInstrumentVA (Item insTemplate, uint32 tag1, ...)
```

Description

This procedure allocates an Instrument based on an instrument Template, previously loaded using `LoadInsTemplate()` or similar call, and allocates the DSP resources necessary for the Instrument. See also the convenience function `LoadInstrument()` combines these two steps.

The new instrument is always created in the `AF_STOPPED` state, regardless of the setting of `AF_INSF_ABANDONED`.

Call `DeleteInstrument()` when you're finished with the Instrument to deallocate its resources.

Arguments

`insTemplate`

Item number of an Instrument Template from which to allocate the Instrument.

Tag Arguments

`AF_TAG_CALCRATE_DIVIDE (uint32)`

Specifies the denominator of the fraction of the total DSP cycles on which this instrument is to run. The valid settings for this are:

1 - Full rate execution (44,100 cycles/sec)

2 - Half rate (22,050 cycles/sec)

8 - 1/8 rate (5,512.5 cycles/sec) (new for M2)

Defaults to 1.

`AF_TAG_PRIORITY (uint8)`

Priority of new instrument in range of 0..200. Defaults to 100.

`AF_TAG_SET_FLAGS (uint32)`

Set of `AF_INSF_` flags to set in new instrument. All flags default to cleared. See the Instrument item page a for complete description of `AF_INSF_` flags.

Return Value

The procedure returns the item number for the Instrument (a positive value) or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in `libc.a V29`.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_INSTRUMENT_NODE),  
              AF_TAG_TEMPLATE, insTemplate,  
              TAG_JUMP, tagList);
```

Associated Files

<:audio:audio.h>, libc.a

See Also

DeleteInstrument(), LoadInsTemplate(), LoadInstrument(), Instrument

DeleteInstrument

Frees an Instrument allocated by CreateInstrument().

Synopsis

```
Err DeleteInstrument (Item InstrumentItem)
```

Description

This procedure frees an instrument allocated by CreateInstrument(), which frees the resources allocated for the instrument. If the instrument is running, it is stopped. All of the Knobs and Probes belonging to this Instrument are deleted. All Attachments made to this Instrument are deleted. If any of those attachments were created with { AF_TAG_AUTO_DELETE_SLAVE, TRUE }, the associated slave items are also deleted. All connections to this Instrument are broken.

If you delete an instrument Template, all Instruments created using that Template are freed, so you don't need to use DeleteInstrument() on them.

Do not confuse this function with UnloadInstrument(). If you create an Instrument using LoadInstrument(), you shouldn't use DeleteInstrument() to free the instrument because it will leave the instrument's Template loaded and without any handle to the Template with which to unload it. Balance CreateInstrument() with DeleteInstrument(); LoadInstrument() with UnloadInstrument().

Arguments

InstrumentItem

The item number of the instrument to free.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro in <:audio:audio.h>

Associated Files

<:audio:audio.h>

See Also

CreateInstrument(), Instrument, Attachment

DisconnectInstrumentParts

Breaks a connection made by
`ConnectInstrumentParts()`.

Synopsis

```
Err DisconnectInstrumentParts (  
    Item dstInstrument, const char *dstPortName, int32 dstPartNum)
```

Description

This procedure breaks a connection made by `ConnectInstrumentParts()` between two instruments. If the connection was to a knob of the second instrument, the knob is once again available for tweaking. Since only one connection can be made to any given destination, it is not necessary to specify the Source.

Arguments

`dstInstrument`

Item number of the destination instrument.

`dstPortName`

Name of the input port of the destination instrument to break connection to.

`dstPartNum`

Part index of output port of the destination instrument to connect to. For ports with a single part, use zero.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V27.

Associated Files

`<:audio:audio.h>`

See Also

`ConnectInstrumentParts()`

DumpInstrumentResourceInfo

Prints out instrument resource usage information.

Synopsis

```
void DumpInstrumentResourceInfo (Item insOrTemplate, const char
*banner)
```

Description

Prints all resource usage information for the specified instrument or template to the debug console.

Arguments

insOrTemplate

Instrument or Template to print information for.

banner

Description of print out. Can be NULL.

Implementation

Library call implemented in libc.a V30.

Associated Files

<:audio:audio.h>, libc.a

See Also

GetInstrumentResourceInfo()

GetInstrumentPortInfoByIndex

Looks up Instrument port by index and returns port information.

Synopsis

```
Err GetInstrumentPortInfoByIndex (InstrumentPortInfo *info, uint32
infoSize,
                                Item insOrTemplate, uint32
portIndex)
```

Description

This function looks up an instrument port by index and returns information about the port. The information is returned in an InstrumentPortInfo structure.

The InstrumentPortInfo structure contains the following fields:

pinfo_Name
Name of the port.

pinfo_Type
Port type of the port (a AF_PORT_TYPE_ value).

pinfo_SignalType
Signal type of the port (AF_SIGNAL_TYPE_). This only applies to the following port types: AF_PORT_TYPE_INPUT, AF_PORT_TYPE_OUTPUT, AF_PORT_TYPE_KNOB, and AF_PORT_TYPE_ENVELOPE.

pinfo_NumParts
Number of parts that the port has. This is in the range of 1..N. Part numbers used to refer to this port are in the range of 0..N-1. This only applies to the following port types: AF_PORT_TYPE_INPUT, AF_PORT_TYPE_OUTPUT, and AF_PORT_TYPE_KNOB.

This function in combination with GetNumInstrumentPorts() permits one to walk the port list of an instrument.

Arguments

info
A pointer to an InstrumentPortInfo structure where the information will be stored.

infoSize
The size in bytes of the InstrumentPortInfo structure.

insOrTemplate
Instrument or Instrument Template Item to query.

portIndex
Index of the port in the range of 0..numPorts-1, where numPorts is returned by GetNumInstrumentPorts().

Return Value

Non-negative if successful or an error code (a negative value) if an error occurs. On failure, the contents of info are left unchanged.

Implementation

Folio call implemented in audio folio V30.

Example

```
// display instrument port info
void DumpInstrumentPortInfo (Item insOrTemplate)
{
    InstrumentPortInfo info;
    const int32 numPorts = GetNumInstrumentPorts (insOrTemplate);
    int32 i;

    printf ("Instrument 0x%x has %d port(s)\n", insOrTemplate,
numPorts);
    for (i=0; i<numPorts; i++) {
        if (GetInstrumentPortInfoByIndex (&info, sizeof info,
insOrTemplate, i) >= 0) {
            printf ("%2d: '%s' port type=%d parts=%d signal
type=%d\n",
                i, info.pinfo_Name, info.pinfo_Type,
info.pinfo_NumParts,
                info.pinfo_SignalType);
        }
    }
}
```

Caveats

Because envelope hooks also can be also be treated as knobs, an AF_PORT_TYPE_ENVELOPE port will also have an associated pair of AF_PORT_TYPE_KNOBs. For example, envelope.dsp has the following envelope related ports:

Env - AF_PORT_TYPE_ENVELOPE

Env.incr - AF_PORT_TYPE_KNOB

Env.request - AF_PORT_TYPE_KNOB

Associated Files

<:audio:audio.h>

See Also

GetInstrumentPortInfoByName(),
GetAudioSignalInfo(), insinfo

GetNumInstrumentPorts(),

GetInstrumentPortInfoByName

Looks up Instrument port by name and returns port information.

Synopsis

```
GetInstrumentPortInfoByName (InstrumentPortInfo *info, uint32
infoSize,
                                Item insOrTemplate, const char
*portName)
```

Description

This function looks up an instrument port by name and returns information about the port. The information is returned in an InstrumentPortInfo structure.

The InstrumentPortInfo structure contains the following fields:

pinfo_Name
Name of the port.

pinfo_Type
Port type of the port (a AF_PORT_TYPE_ value).

pinfo_SignalType
Signal type of the port (AF_SIGNAL_TYPE_). This only applies to the following port types: AF_PORT_TYPE_INPUT, AF_PORT_TYPE_OUTPUT, AF_PORT_TYPE_KNOB, and AF_PORT_TYPE_ENVELOPE.

pinfo_NumParts
Number of parts that the port has. This is in the range of 1..N. Part numbers used to refer to this port are in the range of 0..N-1. This only applies to the following port types: AF_PORT_TYPE_INPUT, AF_PORT_TYPE_OUTPUT, and AF_PORT_TYPE_KNOB.

Arguments

info
A pointer to an InstrumentPortInfo structure where the information will be stored.

infoSize
The size in bytes of the InstrumentPortInfo structure.

insOrTemplate
Instrument or Instrument Template Item to query.

portName
Name of the port to query. Port names are matched case-insensitively.

Return Value

Non-negative if successful or an error code (a negative value) if an error occurs. On failure, the contents of info are left unchanged.

Implementation

Folio call implemented in audio folio V30.

Caveats

Because envelope hooks also can be also be treated as knobs, an `AF_PORT_TYPE_ENVELOPE` port will also have an associated pair of `AF_PORT_TYPE_KNOBS`. For example, `envelope.dsp` has the following envelope related ports:

`Env` - `AF_PORT_TYPE_ENVELOPE`

`Env.incr` - `AF_PORT_TYPE_KNOB`

`Env.request` - `AF_PORT_TYPE_KNOB`

Associated Files

`<:audio:audio.h>`

See Also

`GetInstrumentPortInfoByIndex()`,
`GetAudioSignalInfo()`, `insinfo`

`GetNumInstrumentPorts()`,

GetInstrumentResourceInfo

Get audio resource usage information for an Instrument.

Synopsis

```
Err GetInstrumentResourceInfo (InstrumentResourceInfo *info,  
                               uint32 infoSize, Item insOrTemplate,  
                               uint32 rsrcType)
```

Description

This function returns information about the resource usage of the supplied Instrument, or any Instrument created from the supplied Instrument Template. The information is returned in an InstrumentResourceInfo structure.

The InstrumentResourceInfo structure contains the following fields:

`rinfo_PerInstrument`

Indicates the amount of the resource required for each instrument created from the same template.

`rinfo_MaxOverhead`

Indicates the worst case overhead shared among all instruments from the same template. This typically represents the size of shared code or data used by the instrument.

A shared resource is allocated when the first instrument which uses it is created, and freed when the last instrument which uses it is deleted. Many different templates can share the same shared resource. Because of this, less than this amount may actually be required to allocate the first instrument from this template.

The total amount of a resource required by all instruments from a template (worst case) is computed with the formula:

$$\text{rinfo_MaxOverhead} + \text{rinfo_PerInstrument} * \text{<number of instruments>}$$

Arguments

`info`

A pointer to an InstrumentResourceInfo structure where the information will be stored.

`infoSize`

The size in bytes of the InstrumentResourceInfo structure.

`insOrTemplate`

Instrument or Instrument Template Item to query.

`rsrcType`

The resource type to query. This must be one of the AF_RESOURCE_TYPE_ values defined in `<:audio:audio.h>`. See below.

Resource Types

`AF_RESOURCE_TYPE_TICKS`

Number of DSP ticks / frame that instrument uses.

`AF_RESOURCE_TYPE_CODE_MEM`

Words of DSP code memory that instrument uses.

AF_RESOURCE_TYPE_DATA_MEM

Words of DSP data memory that instrument uses.

AF_RESOURCE_TYPE_FIFOS

Number of FIFOs (input and output counted together) that instrument uses.

AF_RESOURCE_TYPE_TRIGGERS

Number of triggers that instrument uses.

Return Value

Non-negative if successful or an error code (a negative value) if an error occurs. On failure, the contents of info are left unchanged.

Implementation

Folio call implemented in audio folio V30.

Caveats

Tick amounts returned by this function are in ticks per frame regardless of the instrument's AF_TAG_CALCULATE_DIVIDE setting. Tick amounts returned by GetAudioResourceInfo() are in ticks per batch (8 frame set). So an instrument created with full rate execution will reduce the available ticks reported by GetAudioResourceInfo() by eight times the amount returned by GetInstrumentResourceInfo().

Associated Files

<:audio:audio.h>

See Also

DumpInstrumentResourceInfo(), insinfo, GetAudioResourceInfo()

GetNumInstrumentPorts

Returns the number of ports of an Instrument.

Synopsis

```
int32 GetNumInstrumentPorts (Item insOrTemplate)
```

Description

This function returns the number of ports of an Instrument or instrument Template. This function in combination with `GetInstrumentPortInfoByIndex()` permits one to walk the port list of an instrument.

Arguments

`insOrTemplate`
Instrument or Instrument Template Item to query.

Return Value

Non-negative value indicating the number of ports belonging to the instrument, or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V30.

Associated Files

<:audio:audio.h>

See Also

`GetInstrumentPortInfoByIndex()`, `GetInstrumentResourceInfo()`

LoadInsTemplate

Loads a standard DSP Instrument Template.

Synopsis

```
Item LoadInsTemplate ( const char *insName, const TagArg *tagList )
```

```
Item LoadInsTemplateVA ( const char *insName, uint32 tag1, ... )
```

Description

This procedure loads an Instrument Template from the specified file. Note that the procedure doesn't create an instrument from the instrument template. Call `CreateInstrument()` to create an Instrument from this Template.

When you finish using the Template, call `UnloadInsTemplate()` to deallocate the template's resources.

Arguments

`insName`
Name of the instrument, e.g., `sawtooth.dsp`.

`tagList`
Tag list. None currently supported. Pass `NULL`.

Return Value

The procedure returns a Template item number if successful (a positive value) or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

`<:audio:audio.h>`, `libc.a`

See Also

`UnloadInsTemplate()`, `CreateMixerTemplate()`, `CreatePatchTemplate()`,
`CreateInstrument()`, `LoadInstrument()`, `CreateAttachment()`,
`LoadPatchTemplate()`, `LoadScoreTemplate()`

LoadInstrument

Loads a DSP instrument Template and creates an Instrument from it in one call.

Synopsis

```
Item LoadInstrument (const char *name, uint8 calcRateDivider,  
                    uint8 priority)
```

Description

This functions combines the actions of LoadInsTemplate() and CreateInstrument(), and returns the resulting Instrument item.

Call UnloadInstrument() (not DeleteInstrument()) to free an Instrument created by this function. Calling DeleteInstrument() deletes the Instrument, but not the Template, leaving you with an unaccessible Template Item that you can't delete.

Arguments

name

Name of the file containing the instrument template.

calcRateDivider

Specifies the denominator of the fraction of the total DSP cycles on which this instrument is to run. The valid settings for this are:

1 - Full rate execution (44,100 cycles/sec)

2 - Half rate (22,050 cycles/sec)

8 - 1/8 rate (5,512.5 cycles/sec) (new for M2)

Zero is treated as one.

priority

Determines order of execution in DSP. Set from 0 to 200. A typical value would be 100. This also determines the priority over other instruments when voices are stolen for dynamic voice allocation.

Return Value

The procedure returns an Instrument item number (a positive value) if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in audio folio V20.

Caveats

This function creates a separate Template Item for each Instrument it creates. If you intend to create multiple Instruments of the same type, it is more efficient to use LoadInsTemplate() to create the Template once and multiple calls to CreateInstrument().

This function can only be used to load standard DSP instrument templates; it can't load ARIA patches or create mixers.

Associated Files

`<:audio:audio.h>`

See Also

`UnloadInstrument()`, `LoadInsTemplate()`, `CreateInstrument()`

PauseInstrument

Pauses an Instrument's playback.

Synopsis

```
Err PauseInstrument (Item Instrument)
```

Description

This procedure pauses an instrument during playback. A paused instrument ceases to play, but retains its position in playback so that it can resume playback at that point. `ResumeInstrument()` allows a paused instrument to continue its playback.

This procedure is intended primarily for sampled-sound instruments, where a paused instrument retains its playback position within a sampled sound. `PauseInstrument()` and `ResumeInstrument()` used with sound-synthesis instruments may not have effects any different from `StartInstrument()` and `StopInstrument()`.

The paused state is superceded by a call to `StartInstrument()` or `StopInstrument()`.

!!! unsure whether `ReleaseInstrument()` affects a paused instrument.

Arguments

Instrument

The item number for the instrument.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>

Caveats

This procedure will not pause an envelope attached to an instrument. The envelope will continue to play while the instrument is paused.

See Also

`ResumeInstrument()`,
`ReleaseInstrument()`

`StartInstrument()`,

`StopInstrument()`,

ReleaseInstrument

Instruct an Instrument to begin to finish (Note Off).

Synopsis

```
Err ReleaseInstrument (Item Instrument, const TagArg *tagList)
```

```
Err ReleaseInstrumentVA (Item Instrument, uint32 tag1, ...)
```

Description

This tells an Instrument to progress into its "release phase" if it hasn't already done so. This is equivalent to a MIDI Note Off event. Any Samples or Envelopes that are attached are set to their release portion, which may or may not involve a release loop. The sound may continue to be produced indefinitely depending on the release characteristics of the instrument.

This has no audible effect on instruments that don't have some kind of sustain loop (e.g. `sawtooth.dsp`).

Affects only instruments in the `AF_STARTED` state: sets them to the `AF_RELEASED` state. By default if and when an instrument reaches the end of its release phase, it stays in the `AF_RELEASED` state until explicitly changed (e.g. `StartInstrument()`, `StopInstrument()`). If one of this Instrument's running Attachments has the `AF_ATTFFATLADYSINGS` flag set, the Instrument is automatically stopped when that Attachment completes. In that case, the instrument goes into the `AF_STOPPED` state, or, if the Instrument has the `AF_INSF_AUTOABANDON` flag set, the `AF_ABANDONED` state.

!!! not sure of the effect this has on a paused instrument

Arguments

Instrument

The item number for the instrument.

Tag Arguments

`AF_TAG_TIME_SCALE_FP` (float32)

Scales times for all Envelopes attached to this Instrument which do not have the `AF_ENVF_LOCKTIMESCALE` flag set. Defaults to value set by `StartInstrument()`.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

`StartInstrument()`, `StopInstrument()`, `PauseInstrument()`,
`ResumeInstrument()`, `Instrument`

ResumeInstrument

Resumes playback of a paused instrument.

Synopsis

```
Err ResumeInstrument (Item Instrument)
```

Description

This procedure resumes playback of an instrument paused using `PauseInstrument()`. A resumed instrument continues playback from the point where it was paused. It does not restart from the beginning of a note.

This procedure is intended primarily for sampled-sound instruments, where a paused instrument retains its playback position within a sampled sound. `PauseInstrument()` and `ResumeInstrument()` used with sound-synthesis instruments may not have effects any different than `StartInstrument()` and `StopInstrument()`.

This function has no effect on an instrument that has been stopped or restarted after being paused.

Arguments

Instrument

The item number for the instrument.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>

See Also

`PauseInstrument()`,
`ReleaseInstrument()`

`StartInstrument()`,

`StopInstrument()`,

StartInstrument

Begins playing an Instrument (Note On).

Synopsis

```
Err StartInstrument (Item Instrument, const TagArg *tagList)
```

```
Err StartInstrumentVA (Item Instrument, uint32 tag1, ...)
```

Description

This procedure begins execution of an instrument. This typically starts a sound but may have other results, depending on the nature of the instrument. This call links the DSP code into the list of active instruments. If the instrument has Samples or Envelopes attached, they will also be started (unless the Attachments specify otherwise). This is equivalent to a MIDI "Note On" event.

The Amplitude and Frequency (or SampleRate) knobs, of instruments that have such, can be adjusted by some of the tags listed below before the instrument is started. When none of the tags for a particular knob are specified, that knob is left set to its previous value. At most one tag for each knob can be specified. Tags are ignored for Instruments without the corresponding knob. Knobs connected to the output of another Instrument (see `ConnectInstrumentParts()`), cannot be set in this manner. A Knob that has been grabbed by `CreateKnob()`, can however be set in this manner.

This function puts the instrument in the `AF_STARTED` state. If the instrument was previous running, it is first stopped and then restarted. If the instrument has a sustain or release loop, it stays in the `AF_STARTED` state until the state is explicitly changed (e.g. `ReleaseInstrument()`, `StopInstrument()`).

This function supercedes a call to `PauseInstrument()`.

Arguments

Instrument

The item number for the instrument.

Tag Arguments

Amplitude:

These tags apply to instruments that have an unconnected Amplitude knob. They are mutually exclusive except for `AF_TAG_EXP_VELOCITY_SCALAR`. All MIDI velocity tags also perform multi-sample selection based on Sample velocity ranges.

`AF_TAG_AMPLITUDE_FP` (float32) - Start

Value to set instrument's Amplitude knob to before starting instrument (for instruments that have an Amplitude knob). Valid range -1.0..1.0.

`AF_TAG_VELOCITY` (uint8) - Start

Linear MIDI key velocity to amplitude mapping:

$\text{Amplitude} = \text{Velocity} / 127.0$

MIDI key velocity is specified in the range of 0..127.

`AF_TAG_SQUARE_VELOCITY` (uint8) - Start

Like `AF_TAG_VELOCITY` except:

$\text{Amplitude} = (\text{Velocity}/127.0)**2.$

This gives a more natural response curve than the linear version.

AF_TAG_EXPONENTIAL_VELOCITY (uint8) - Start

Like AF_TAG_VELOCITY except:

$\text{Amplitude} = 2.0**((\text{Velocity}-127)*\text{expVelocityScalar}).$

where expVelocityScalar is set using AF_TAG_EXP_VELOCITY_SCALAR. This gives a more natural response curve than the linear version.

AF_TAG_EXP_VELOCITY_SCALAR (float32) - Start

Sets expVelocityScalar used by AF_TAG_EXPONENTIAL_VELOCITY. These two tags are used together with StartInstrument() and can be in either order. The default value is (1.0/20.0), which means that the Amplitude will double for every 20 units of velocity.

Frequency:

These tags apply to instruments that support frequency settings (e.g., oscillators, LFOs, sample players). They have no effect if the Frequency or SampleRate knob is connected to another instrument. They are mutually exclusive.

AF_TAG_FREQUENCY_FP (float32)

Play instrument at a specific output frequency.

Oscillator or LFO: Sets the Frequency knob to the specified value.

Sample player with a tuned sample: Adjusts the SampleRate knob to play the sample at the desired frequency. For example, to play sinewave.aiff at 440Hz, set this tag to 440.

AF_TAG_PITCH (uint8)

MIDI note number of the pitch to play instrument at. The range is 0 to 127; 60 is middle C. Once the output frequency is determined applies the frequency in the same manner as AF_TAG_FREQUENCY_FP.

This tag also performs multi-sample selection based on Sample note ranges, and Envelope pitch-based time scaling.

AF_TAG_RATE_FP (float32)

Set raw frequency control. For samplers, this is a fraction of DAC rate. For oscillators, this is a fractional phase increment.

AF_TAG_SAMPLE_RATE_FP (float32)

Sample rate in Hz to set sample player's SampleRate knob to.

AF_TAG_DETUNE_FP (float32)

For samplers, play at a fraction of original recorded sample rate. For oscillators or LFOs, play at a fraction of default frequency.

Time Scaling:

AF_TAG_TIME_SCALE_FP (float32)

Scales times for all Envelopes attached to this Instrument which do not have the AF_ENVF_LOCKTIMESCALE flag set, after pitch-based time scaling is performed. Defaults to 1.0. This setting remains in effect until the instrument is started again or is changed by ReleaseInstrument().

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<.:audio:audio.h>

See Also

ReleaseInstrument(), StopInstrument(), PauseInstrument(),
ResumeInstrument(), Instrument, Sample

StopInstrument

Abruptly stops an Instrument.

Synopsis

```
Err StopInstrument (Item Instrument, const TagArg *tagList)
```

```
Err StopInstrumentVA (Item Instrument, uint32 tag1, ...)
```

Description

This procedure, which abruptly stops an instrument, is called when you want to abort the execution of an instrument immediately. This can cause a click because of its suddenness. You should use `ReleaseInstrument()` to gently release an instrument according to its release characteristics.

Affects only instruments in the `AF_STARTED` or `AF_RELEASED` states: sets them to the `AF_STOPPED` state, or, if the Instrument has the `AF_INSF_AUTOABANDON` flag set, the `AF_ABANDONED` state.

This function supercedes a call to `PauseInstrument()`.

Arguments

Instrument

The item number for the instrument.

Tag Arguments

None

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

`StartInstrument()`, `ReleaseInstrument()`, `PauseInstrument()`,
`ResumeInstrument()`, `Instrument`

UnloadInsTemplate

Unloads an instrument Template loaded by LoadInsTemplate().

Synopsis

```
Err UnloadInsTemplate (Item insTemplate)
```

Description

This procedure unloads (or deletes) an instrument template created by LoadInsTemplate(). It unloads the Template from memory and frees its resources. All Instruments created using the Template are deleted, with the side effects of DeleteInstrument(). All Attachments made to this Template are deleted. If any of those attachments were created with { AF_TAG_AUTO_DELETE_SLAVE, TRUE }, the associated slave items are also deleted.

Arguments

insTemplate
Item number of Template to delete.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:audio.h> V29.

Associated Files

<:audio:audio.h>

See Also

LoadInsTemplate(), Template, Instrument, Attachment

UnloadInstrument

Unloads an instrument loaded with
`LoadInstrument()`.

Synopsis

```
Err UnloadInstrument (Item instrument)
```

Description

This procedure frees the Instrument and unloads the Template loaded by `LoadInstrument()`. This has the same side effects as `DeleteInstrument()` and `UnloadInsTemplate()`.

Do not confuse this function with `DeleteInstrument()`, which deletes an Instrument created by `CreateInstrument()`. Calling `DeleteInstrument()` for an instrument created by `LoadInstrument()` deletes the Instrument, but not the Template, leaving you with an unaccessible Template Item that you can't delete. Calling `UnloadInstrument()` for an Instrument created by `CreateInstrument()` deletes Template for that Instrument along with all other Instruments created from that Template.

Arguments

instrument
Item number of the instrument.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>

See Also

`LoadInstrument()`, `DeleteInstrument()`, `UnloadInsTemplate()`

CreateKnob

Gain direct access to one of an Instrument's Knobs.

Synopsis

```
Item CreateKnob (Item instrument, const char *name, const TagArg
*tagList)
```

```
Item CreateKnobVA (Item instrument, const char *name, uint32 tag1,
...)
```

Description

This procedure creates a Knob item that provides a fast connection between a task and one of an Instrument's parameters. You can then call `SetKnobPart()` to rapidly modify that parameter.

See the Instrument Template pages for complete listings of each Instrument Templates knobs.

Call `DeleteKnob()` to relinquish access to this knob. All Knob Items grabbed for an Instrument are deleted when the Instrument is deleted. This can save you a bunch of calls to `DeleteKnob()`.

Arguments

instrument

The item number of the Instrument.

name

The name of the knob to grab. The knob name is matched case-insensitively.

Tag Arguments

AF_TAG_TYPE (uint8)

Determines the signal type of the knob. Must be one of the `AF_SIGNAL_TYPE_*` defined in `<:audio:audio.h>`. Defaults to the default signal type of the knob (for standard DSP instruments, this is described in the instrument documentation).

Return Value

The procedure returns the item number of the Knob (a positive value) if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in `libc.a V29`.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_KNOB_NODE),
              AF_TAG_INSTRUMENT, instrument,
              AF_TAG_NAME, name,
              TAG_JUMP, tagList);
```

Associated Files

`<:audio:audio.h>`, `libc.a`

See Also

DeleteKnob(), SetKnobPart(), Knob

DeleteKnob

Deletes a Knob.

Synopsis

```
Err DeleteKnob (Item knob)
```

Description

This procedure deletes the Knob Item.

Arguments

knob
Item number of knob to delete.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>`.

Associated Files

`<:audio:audio.h>`

See Also

CreateKnob()

ReadKnob

Reads the current value of a single-part Knob.

Synopsis

```
Err ReadKnob (Item knob, float32 *valuePtr)
```

Description

This procedure reads the current value of a single-part knob.

Arguments

knob

Item number of the knob to be read. Always reads part 0 of the knob.

valuePtr

Pointer to a variable to receive the value for that knob. Task must have write permission for that address. The result is in the units that apply to the `AF_SIGNAL_TYPE_` of the Knob.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience macro implemented in `<:audio:audio.h>` V27.

Notes

This macro is equivalent to:

```
ReadKnobPart (knob, 0, valuePtr)
```

Associated Files

`<:audio:audio.h>`

See Also

`SetKnobPart()`, `CreateKnob()`, `ReadKnobPart()`, `Knob`

ReadKnobPart

Reads the current value of a Knob.

Synopsis

```
Err ReadKnobPart (Item knob, int32 partNum, float32 *valuePtr)
```

Description

This procedure reads the current value of a knob. This is either the last value set by `SetKnobPart()`, or the default knob value if `SetKnobPart()` hasn't been called for this knob yet.

In the case of a knob which is connected to another instrument (via `ConnectInstrumentParts()`), this function still returns the last set value, not the signal being injected into this knob via the connection. To read the signal value use a Probe on the source instrument's output.

Arguments

knob

Item number of the knob to be read.

partNum

Index of the part of the knob to be read.

valuePtr

Pointer to a variable to receive the value for that knob. Task must have write permission for that address. The result is in the units that apply to the `AF_SIGNAL_TYPE_` of the Knob.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V27.

Associated Files

<:audio:audio.h>, libc.a

See Also

`SetKnobPart()`, `CreateKnob()`, `ReadKnob()`, Knob

SetKnob

Sets the value of a single-part Knob.

Synopsis

```
Err SetKnob (Item knob, float32 value)
```

Description

This procedure sets the value of a single-part knob.

Arguments

knob

Item number of the knob to be set. Always sets part 0.

value

The new value for that knob. This value is in the units apply to the AF_SIGNAL_TYPE_ of the Knob.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience macro implemented in `<:audio:audio.h>` V27.

Notes

This macro is equivalent to:

```
SetKnobPart (knob, 0, value)
```

Associated Files

`<:audio:audio.h>`

See Also

`ReadKnobPart()`, `CreateKnob()`, `SetKnobPart()`, `Knob`

SetKnobPart

Sets the value of a Knob.

Synopsis

```
Err SetKnobPart (Item knob, int32 partNum, float32 value)
```

Description

This procedure sets the value of a knob. This has been optimized to allow rapid modulation of sound parameters. The value is clipped to the allowable range and is converted to the appropriate internal units as necessary.

The current value of the Knob can be read by calling `ReadKnobPart()`.

If this knob is connected to the output of another instrument via `ConnectInstrumentParts()`, the set value is ignored until that connection is broken.

Arguments

knob

Item number of the knob to be set.

partNum

Index of the part of the knob to be set.

value

The new value for that knob. This value is in the units that apply to the `AF_SIGNAL_TYPE_` of the Knob.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V27.

Associated Files

`<.:audio:audio.h>`, `libc.a`

See Also

`ReadKnobPart()`, `CreateKnob()`, `SetKnob()`, `Knob`

CloseAudioFolio

Closes the audio folio.

Synopsis

```
Err CloseAudioFolio (void)
```

Description

This procedure closes a task's connection with the audio folio. Call it when your application finishes using audio calls.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:audio.h>, libc.a

See Also

OpenAudioFolio()

EnableAudioInput

Enables or disables audio input.

Synopsis

```
Err EnableAudioInput (int32 OnOrOff, const TagArg *tagList)
```

```
Err EnableAudioInputVA (int32 OnOrOff, uint32 tag1, ...)
```

Description

Enables or disables the use of audio input for the calling task. The DSP instrument `line_in.dsp` requires that a successful call to `EnableAudioInput()` be made before that instrument can be loaded.

Enable/Disable calls nest. Enabling audio input is tracked for each task; you don't need disable it as part of your cleanup.

Arguments

OnOrOff

Non-zero to enable, zero to disable. An enable count is maintained for each task so that you may nest calls to `EnableAudioInput(TRUE, ...)`.

Tag Arguments

None

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V24.

Caveats

The use of audio input may require a special licensing agreement.

`EnableAudioInput()` only permits a NULL tag list pointer. Therefore, `EnableAudioInputVA()` always fails with `AF_ERR_BADTAG`.

Associated Files

<:audio:audio.h>, libc.a

See Also

`line_in.dsp`

GetAudioFolioInfo

Get system-wide audio settings.

Synopsis

```
Err GetAudioFolioInfo (TagArg *tagList)
```

Description

Queries audio folio for certain system-wide settings. It takes a tag list with the `ta_Tag` fields set to parameters to query and fills in the `ta_Arg` field of each `TagArg` with the parameter requested by that `TagArg`.

For normal multiplayers, these settings are set to the described normal values during startup. Other kinds of hardware or execution environments (e.g. cable) may have different settings.

None of these paramaters can be set directly by an application.

Arguments

`tagList`

Pointer to tag list containing `AF_TAG_` tags to query. The `ta_Arg` fields of each matched tag is filled in with the corresponding setting from the audio folio. `TAG_JUMP` and `TAG_NOP` are treated as they normally are for tag processing.

Tag Arguments

`AF_TAG_SAMPLE_RATE_FP` (float32)

Returns the audio DAC sample rate in samples/second expressed as a float32 value. Normally this is 44,100 samples/second. For certain cable applications this can be 48,000 samples/second.

When the sample rate is something other than 44,100, substitute this to everywhere in the audio documentation that specifies the sample rate as 44,100 samples/second. If your application needs to cope with variable DAC sample rates and you have need the sample rate for something, then you should read this value from here rather than hardcoding 44,100, or any other sample rate, into your code.

`AF_TAG_AMPLITUDE_FP` (float32)

Returns the audio folio's master volume level. Normally this is 1.0.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

`AF_ERR_BADTAG`

If this function is passed any tag that is not listed above.

Additional Results

Fills in the `ta_Arg` field of each matched `TagArg` in `tagList` with the corresponding setting from the audio folio.

Implementation

Folio call implemented in audio folio V24.

Associated Files

<:audio:audio.h>, libc.a

See Also

GetAudioItemInfo()

GetAudioFrameCount

Gets count of audio frames executed.

Synopsis

```
uint32 GetAudioFrameCount (void)
```

Description

Gets 32-bit value that is incremented on every DSP sample frame, typically at 44,100 Hz. This is only useful for determining relative elapsed frame counts.

Return Value

32-bit sample frame count.

Implementation

Folio call implemented in audio folio V24.

Associated Files

<:audio:audio.h>, libc.a

Example

```
NewFrameCount = GetAudioFrameCount();  
ElapsedFrames = NewFrameCount - OldFrameCount;  
OldFrameCount = NewFrameCount;
```

Caveats

The DSP executes asynchronously from the DAC. The fact, therefore, that the DSP frame counter has advanced by N samples does not imply that the DAC has necessarily played exactly N samples. The DSP processes samples in short bursts. GetAudioFrameCount() should, therefore, not be used as a measure of time, but rather a measure of how many frames the DSP has processed.

See Also

GetAudioTime()

GetAudioItemInfo

Gets information about an audio item.

Synopsis

```
Err GetAudioItemInfo (Item audioItem, TagArg *tagList)
```

Description

This procedure queries settings of several kinds of audio items. It takes a tag list with the `ta_Tag` fields set to parameters to query and fills in the `ta_Arg` field of each `TagArg` with the parameter requested by that `TagArg`.

-- `SetAudioItemInfo()` can be used to set audio item parameters.

Arguments

`audioItem`

Item number of an audio item to query.

`tagList`

Pointer to tag list containing `AF_TAG_` tags to query. The `ta_Arg` fields of each matched tag is filled in with the corresponding setting from the audio item. `TAG_JUMP` and `TAG_NOP` are treated as they normally are for tag processing.

See each audio Item pages for list of tags that are supported by each item type.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

`AF_ERR_BADTAG`

If this function is passed any tag that is not listed in the appropriate item page as supporting being queried.

`AF_ERR_BADITEM`

If this function is passed an Item of a type which it doesn't support.

Fills in the `ta_Arg` field of each matched `TagArg` in `tagList` with the corresponding setting from the audio item. If a bad tag is passed, all `TagArgs` prior to the bad tag are filled out.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

`SetAudioItemInfo()`, `Attachment`, `AudioClock`, `Cue`, `Envelope`, `Instrument`, `Knob`, `Probe`, `Sample`, `Template`, `Tuning`

GetAudioResourceInfo

Get information about available audio resources.

Synopsis

```
Err GetAudioResourceInfo (AudioResourceInfo *info, uint32 infoSize,  
                          uint32 rsrcType)
```

Description

This function returns availability information of a specified audio resource. The information is returned in an AudioResourceInfo structure.

The AudioResourceInfo structure contains the following fields:

rinfo_Total
Indicates the total amount of the resource in the system.

rinfo_Free
Indicates the amount of the resource currently available in the system.

rinfo_LargestFreeSpan
Indicates the size of the largest contiguous block of the resource. This only applies to memory resources (AF_RESOURCE_TYPE_CODE_MEM and AF_RESOURCE_TYPE_DATA_MEM). This field is set to 0 for the others.

Arguments

info
A pointer to an AudioResourceInfo structure where the information will be stored.

infoSize
The size in bytes of the AudioResourceInfo structure.

rsrcType
The resource type to query. This must be one of the AF_RESOURCE_TYPE_ values defined in <:audio:audio.h>. See below.

Resource Types

AF_RESOURCE_TYPE_TICKS
Number of DSP ticks / batch (8 frame set) available.

AF_RESOURCE_TYPE_CODE_MEM
Words of DSP code memory available.

AF_RESOURCE_TYPE_DATA_MEM
Words of DSP data memory available.

AF_RESOURCE_TYPE_FIFOS
Number of FIFOs available.

AF_RESOURCE_TYPE_TRIGGERS

Number of Triggers available.

Return Value

Non-negative if successful or an error code (a negative value) if an error occurs. On failure, the contents of info are left unchanged.

Implementation

Folio call implemented in audio folio V30.

Caveats

The information returned by GetAudioResourceInfo() is inherently flawed, since you are existing in a multitasking environment. Resources can be allocated or freed asynchronous to the operation of the task calling GetAudioResourceInfo().

Tick amounts returned by this function are in ticks per batch (8 frame set). Tick amounts returned by GetInstrumentResourceInfo() are in ticks per frame regardless of the instrument's AF_TAG_CALCRATE_DIVIDE setting. So an instrument created with full rate execution will reduce the available ticks reported by GetAudioResourceInfo() by eight times the amount returned by GetInstrumentResourceInfo().

Associated Files

<:audio:audio.h>

See Also

GetInstrumentResourceInfo()

OpenAudioFolio

Opens the audio folio.

Synopsis

```
Err OpenAudioFolio (void)
```

Description

This procedure connects a task to the audio folio and must be executed by each task before it makes any audio folio calls. If a task makes an audio folio call before it executes `OpenAudioFolio()`, the task fails.

Call `CloseAudioFolio()` to relinquish a task's access to the audio folio.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libc.a` V29.

Associated Files

`<:audio:audio.h>`, `libc.a`

See Also

`CloseAudioFolio()`

SetAudioItemInfo

Sets parameters of an audio item.

Synopsis

```
Err SetAudioItemInfo (Item audioItem, const TagArg *tagList)
```

```
Err SetAudioItemInfoVA (Item audioItem, uint32 tag1, ...)
```

Description

This procedure sets one or more parameters of an audio item. This is the only way to change parameter settings of an audio item. The parameters are determined by the tag arguments.

If this function fails, the resulting state of the Item you were attempting to modify depends on the kind of item:

Envelope, Sample, Tuning

The item is left unchanged if SetAudioItemInfo() fails for any reason.

Attachment

All of the changes prior to the tag which failed take effect.

Arguments

audioItem

Item number of an audio item to modify.

Tag Arguments

See specific Audio Item pages for supported tags.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Caveats

The failure behavior of attachments may be changed to be like the others.

Associated Files

<:audio:audio.h>, libc.a

See Also

GetAudioItemInfo(), Attachment, AudioClock, Cue, Envelope, Instrument, Knob, Probe, Sample, Template, Tuning

CalcMixerGainPart

Returns the mixer gain knob part number for a specified input and output.

Synopsis

```
int32 CalcMixerGainPart (MixerSpec mixerSpec, int32 inChannel, int32
outChannel)
```

Description

Returns the gain knob part number for a given input and output pair using the following formula:

$$\text{GainKnobPartNumber} = \text{outChannel} * \text{MixerSpecToNumIn}(\text{mixerSpec}) + \text{inChannel}$$
Arguments

mixerSpec
MixerSpec for the mixer for which to compute the gain knob part.

inChannel
Input channel number (0..numInputs).

outChannel
Output channel number (0..numOutputs).

Return Value

Gain knob part number.

Implementation

Macro implemented in audio folio V28.

Caveats

This macro doesn't trap bad input values. SetKnobPart() traps out of range part numbers.

Associated Files

<:audio:audio.h>

See Also

MakeMixerSpec(), CreateMixerTemplate()

CreateMixerTemplate

Creates a custom mixer Template.

Synopsis

```
Item CreateMixerTemplate (MixerSpec mixerSpec, const TagArg *tags );
```

```
Item CreateMixerTemplateVA (MixerSpec mixerSpec, uint32 tag1, ... );
```

Description

This function creates a custom, multi-channel Mixer Template. Use the macro `MakeMixerSpec()` to generate the 32-bit packed `MixerSpec` that you pass to this function. You can specify any number of "Input" channels from 1-32, and 1-8 "Output" channels. There are other options as well, see `MakeMixerSpec()` for more details.

The created mixer has multi-part "Gain" knob that is used to control the amount of each Input that goes to each Output. The Gain knob part number for Input A going to Output B for a mixer with NI Inputs is calculated as follows:

$$\text{GainKnobPartNumber} = A + (B * NI)$$

The macro `CalcMixerGainPart()` does this calculation.

All parts of the Gain knob default to 0. The Amplitude knob, if requested, defaults to 1.0.

Call `DeleteMixerTemplate()` when done with this Template.

Arguments

`mixerSpec`
MixerSpec describing the mixer to construct. Returned by `MakeMixerSpec()`.

Tag Arguments

`TAG_ITEM_NAME (const char *)`
Optional name for the newly created template. Defaults to not having a name.

Return Value

The procedure returns an instrument Template Item number (a non-negative value) if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V28.

Associated Files

<:audio:audio.h>, libc.a

See Also

`MakeMixerSpec()`, `Mixer`, `Template`, `CalcMixerGainPart()`,
`DeleteMixerTemplate()`,

LoadInsTemplate(), CreatePatchTemplate(), CreateInstrument()

DeleteMixerTemplate

Deletes a mixer Template created by
CreateMixerTemplate()

Synopsis

```
Err DeleteMixerTemplate (Item mixerTemplate)
```

Description

This macro deletes a mixer Template created by CreateMixerTemplate(). This has the same side effects as UnloadInsTemplate() (e.g., deleting instruments).

Arguments

```
mixerTemplate  
    Mixer Template Item to delete.
```

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>` V28.

Associated Files

`<:audio:audio.h>`

See Also

CreateMixerTemplate()

MakeMixerSpec

Makes a MixerSpec for use with CreateMixerTemplate().

Synopsis

```
MixerSpec MakeMixerSpec (uint8 numInputs, uint8 numOutputs, uint16 flags)
```

Description

Constructs a 32-bit packed MixerSpec to pass to CreateMixerTemplate() and other mixer-related functions. The mixer created from this spec will have: an "Input" port with N parts, an "Output" port with M parts (unless AF_F_MIXER_WITH_LINE_OUT is set), and a "Gain" knob with N*M parts (see CalcMixerGainPart()). If AF_F_MIXER_WITH_AMPLITUDE is set then it will also have an "Amplitude" knob which acts as a master gain control.

```
Output[m] = Amplitude * (Input[0]*Gain[0,m] + Input[1]*Gain[1,m] +  
.... )
```

Arguments

numInputs

Number of "Input" channels from 1-32. (See caveat below!)

numOutputs

Number of "Output" channels from 1-8.

flags

Set of AF_F_MIXER_ flags (defined below) to control mixer construction.

Flags

AF_F_MIXER_WITH_AMPLITUDE

Mixer will have a global Amplitude knob that can be used as a master gain control.

AF_F_MIXER_WITH_LINE_OUT

Mixer will have an internal connection to the global system mixer. If you don't set this flag the mixer has an "Output" port that can be connected to a line_out.dsp to be heard, or to a delay effect, etc. Note: this flag is only valid if numOutputs is 2.

Return Value

Constructed MixerSpec.

Implementation

Macro implemented in audio folio V28.

Caveats

This macro doesn't trap bad input values, so it is possible to construct an invalid MixerSpecs. CreateMixerTemplate() checks the validity of the MixerSpec it is passed.

Note that there is a bug in V28 that causes mixers with only 1 input to fail! As a workaround, set the AF_F_MIXER_WITH_AMPLITUDE flag and it will work.

Associated Files

<:audio:audio.h>

See Also

CreateMixerTemplate(), CalcMixerGainPart(), MixerSpecToNumIn(),
MixerSpecToNumOut(), MixerSpecToFlags()

MixerSpecToFlags

Extracts mixer flags from MixerSpec.

Synopsis

```
uint16 MixerSpecToFlags (MixerSpec mixerSpec)
```

Description

Returns the AF_F_MIXER_ flags specified in the MixerSpec.

Arguments

```
mixerSpec  
    MixerSpec to examine.
```

Return Value

AF_F_MIXER_ flags.

Implementation

Macro implemented in audio folio V28.

Associated Files

<:audio:audio.h>

See Also

MakeMixerSpec(), MixerSpecToNumIn(), MixerSpecToNumOut()

MixerSpecToNumIn

Extracts number of inputs from MixerSpec.

Synopsis

```
uint8 MixerSpecToNumIn (MixerSpec mixerSpec)
```

Description

Returns the number of inputs specified in the MixerSpec.

Arguments

```
mixerSpec  
MixerSpec to examine.
```

Return Value

Number of inputs.

Implementation

Macro implemented in audio folio V28.

Associated Files

```
<:audio:audio.h>
```

See Also

```
MakeMixerSpec(), MixerSpecToNumOut(), MixerSpecToFlags()
```

MixerSpecToNumOut

Extracts number of outputs from MixerSpec.

Synopsis

```
uint8 MixerSpecToNumOut (MixerSpec mixerSpec)
```

Description

Returns the number of outputs specified in the MixerSpec.

Arguments

```
mixerSpec  
    MixerSpec to examine.
```

Return Value

Number of outputs.

Implementation

Macro implemented in audio folio V28.

Associated Files

<:audio:audio.h>

See Also

MakeMixerSpec(), MixerSpecToNumIn(), MixerSpecToFlags()

CreateProbe

Creates a Probe for an output of an Instrument.

Synopsis

```
Item CreateProbe (Item instrument, const char *portName, const
TagArg *tagList)
```

```
Item CreateProbeVA (Item instrument, const char *portName, uint32
tag1, ... )
```

Description

This procedure creates a Probe item that provides a fast connection between a task and one of an instrument's outputs. You can then call ReadProbePart() to poll the value of the probed output. Use DeleteProbe() to delete the Probe when you are finished with it.

Arguments

instrument
The item number of the instrument.

portName
The name of the Output.

Tag Arguments

AF_TAG_TYPE (uint8) - Create, Query, Modify
Determines the signal type of the probe. Must be one of the AF_SIGNAL_TYPE_* defined in <:audio:audio.h>. Defaults to the signal type of the output (for standard DSP instruments, this is described in the instrument documentation).

Return Value

The procedure returns the item number of the probe if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in libc.a V29.

Associated Files

<:audio:audio.h>, libc.a

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_PROBE_NODE),
              AF_TAG_INSTRUMENT, instrument,
              AF_TAG_NAME,      portName,
              TAG_JUMP,         tagList);
```

See Also

ReadProbePart(), ReadProbe(), DeleteProbe(), Probe

DeleteProbe

Deletes a Probe.

Synopsis

```
Err DeleteProbe (Item probe)
```

Description

This procedure deletes a Probe. All Probes to an Instrument are automatically deleted when that Instrument is deleted.

Arguments

```
probe  
    Item number of probe to delete.
```

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro in `<:audio:audio.h>`

Associated Files

`<:audio:audio.h>`

See Also

CreateProbe(), Probe

ReadProbe

Reads the value of a single-part Probe.

Synopsis

```
Err ReadProbe (Item probe, float *valuePtr)
```

Description

Reads the instantaneous value of a single-part probed instrument output.

Arguments

`probe`
Item number of the probe to be read. Always reads part 0.

`valuePtr`
Pointer to where to put result. Task must have write permission for that address. The result is in the units that apply to the `AF_SIGNAL_TYPE_` of the Probe.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience macro implemented in `<:audio:audio.h>` V27.

Notes

This macro is equivalent to:

```
ReadProbePart (probe, 0, valuePtr)
```

Associated Files

`<:audio:audio.h>`

See Also

`CreateProbe()`, `ReadProbePart()`

ReadProbePart

Reads the value of a Probe.

Synopsis

```
Err ReadProbePart (Item probe, int32 partNum, float *valuePtr)
```

Description

Reads the instantaneous value of a probed instrument output.

Arguments

probe

Item number of the probe to be read.

partNum

Index of the part you wish to read.

valuePtr

Pointer to where to put result. Task must have write permission for that address. The result is in the units that apply to the AF_SIGNAL_TYPE_ of the Probe.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V27.

Associated Files

<:audio:audio.h>, libc.a

See Also

CreateProbe(), ReadProbe()

CreateDelayLine

Creates a Delay Line Sample for echoes and reverberations.

Synopsis

```
Item CreateDelayLine (int32 numBytes, int32 numChannels, bool  
ifLoop)
```

Description

This procedure creates a delay line Sample Item for echo, reverberation, and oscilloscope applications. The delay line Item can be attached to any instrument that provides an output FIFO (e.g., delay_f1.dsp). It can be tapped by a matching 16-bit sampler instrument (e.g., sampler_16_f1.dsp).

When finished with the delay line, delete it using DeleteDelayLine().

Arguments

numBytes

The number of bytes in the delay line's sample buffer. Note that this is in bytes, not frames.

numChannels

The number of channels stored in each sample frame.

ifLoop

TRUE if delay line should be treated as a circular buffer. This is typically the case for delay and reverberation applications.

FALSE for cases in which the delay line should be not be overwritten (e.g., dumping DSP results to memory).

This merely controls whether the delay line has a release loop. A release loop is used to make all of the instruments operating on the delay line immune to ReleaseInstrument(). Using a sustain loop instead would complicate the use of delay lines in patches.

Return Value

The procedure returns the item number of the delay line (non-negative), or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in libc.a V29.

Examples

```
{  
    delay = CreateDelayLine (44100*2, 1, TRUE);    // 1 second  
  
    send = LoadInstrument ("delay_f1.dsp", 0, 100);  
}
```

```
tap = LoadInstrument ("sampler_16_f1.dsp", 0, 100);

sendatt = CreateAttachmentVA (send, delay, TAG_END);

tapatt = CreateAttachmentVA (tap, delay,
    AF_TAG_START_AT, 50,      // 44050 samples away from send;
                           // practically 1 second
    TAG_END);

StartInstrument (tap, NULL);
StartInstrument (send, NULL);

// Now, just connect send's Input to thing to be delayed.
// The delayed signal is available at tap's Ouput.
}
```

Note: Because of the non-atomic start of the two instruments, the actual delay time is not predictable. If precise delay times are critical, create a patch out of the delay instrument template, sampler(s), and signal routing instruments.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_SAMPLE_NODE),
    AF_TAG_DELAY_LINE,    numBytes,
    AF_TAG_CHANNELS,      numChannels,
    AF_TAG_RELEASEBEGIN,  ifLoop ? 0 : -1,
                           // convert numBytes to frames
    AF_TAG_RELEASEEND,    ifLoop ? numBytes / (numChannels * 2) : -1,
    TAG_END);
```

Associated Files

<:audio:audio.h>, libc.a

See Also

Sample, DeleteDelayLine(), CreateAttachment(), delay_f1.dsp,
delay_f2.dsp, sampler_16_f1.dsp, sampler_16_f2.dsp

CreateSample

Creates a Sample.

Synopsis

```
Item CreateSample (const TagArg *tagList)
```

```
Item CreateSampleVA (uint32 tag1, ...)
```

Description

Creates a Sample Item which can be attached to an Instrument or Template with an FIFO.

When you are finished with the Sample, call `DeleteSample()`.

Tag Arguments

!!! add descriptions. See Sample for now.

Return Value

The procedure returns the item number of the sample (a non-negative value) or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in `libc.a V29`.

Caveats

There are some important length and loop point alignment issues with samples. See the Caveats section on the Sample Item page for details.

Examples

```
// Create a Sample Item to be filled out later with
SetAudioItemInfo()
sample = CreateSample (NULL);

// Make a Sample Item for raw sample data in memory
// (8-bit monophonic 22KHz sample in this case)
sample = CreateSampleVA (
    AF_TAG_ADDRESS,      data,
    AF_TAG_FRAMES,       NUMFRAMES,
    AF_TAG_CHANNELS,     1,
    AF_TAG_WIDTH,        1,
    AF_TAG_SAMPLE_RATE_FP, ConvertFP_TagData (22050),
    TAG_END);
```

Notes

This function is equivalent to:

```
CreateItem (MKNODEID(AUDIONODE,AUDIO_SAMPLE_NODE), tagList)
```

Associated Files

<:audio:audio.h>, `libc.a`

See Also

`Sample`, `DeleteSample()`, `LoadSample()`, `CreateAttachment()`

DebugSample

Prints Sample information for debugging.

Synopsis

```
Err DebugSample (Item sample)
```

Description

This procedure dumps all available sample information to the 3DO Debugger screen for your delight and edification. This function is merely a stub in the production version of the audio folio.

Arguments

sample
Item number of a sample.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

Sample, CreateSample()

DeleteDelayLine

Deletes a delay line Sample.

Synopsis

```
Err DeleteDelayLine (Item delayLine)
```

Description

This procedure deletes a delay line item and frees its resources. It also deletes any Attachments to the delay line.

Arguments

```
delayLine
```

The item number of the delay line to delete.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro in *<:audio:audio.h>*

Associated Files

<:audio:audio.h>

See Also

Sample, CreateDelayLine(), DeleteAttachment()

DeleteSample

Deletes a Sample.

Synopsis

```
Err DeleteSample (Item sample)
```

Description

This procedure deletes the specified sample, freeing its resources. It also deletes any Attachments to the sample.

If the Sample was created with { AF_TAG_AUTO_FREE_DATA, TRUE }, then the sample data is also be freed when the Sample Item is deleted. Otherwise, the sample data is not freed automatically.

Arguments

sample
Item number of the Sample to delete.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro in <:audio:audio.h>

Associated Files

<:audio:audio.h>

See Also

Sample, CreateSample(), DeleteAttachment()

ConvertAudioSignalToGeneric

Converts audio signal value of specified signal type to generic signal value.

Synopsis

```
float32 ConvertAudioSignalToGeneric (int32 signalType, int32
rateDivide,
                                     float32 signalValue)
```

Description

This function converts a signal value of the specified signal type (e.g., oscillator frequency in Hertz) to a generic signal value (e.g., signed floating point value in the range of -1.0 to 1.0).

Certain signal types are affected by the execution rate division of the associated instrument:

- AF_SIGNAL_TYPE_SAMPLE_RATE
- AF_SIGNAL_TYPE_OSC_FREQ
- AF_SIGNAL_TYPE_LFO_FREQ

(see description of the Instrument AF_TAG_CALC RATE_DIVIDE tag)

This function does no boundary checking or signal value clipping.

Arguments

signalType

The signal type of the signal to convert.

rateDivide

The execution rate denominator for the signal (e.g., 1=full rate, 2=half rate, etc).

signalValue

Signal value (in the units appropriate for signalType) to convert.

Return Value

Generic value in the range of -1.0 to 1.0 (for signed signals) or 0 to 2.0 (for unsigned signals). Returns 0 when an undefined signalType or rateDivide of 0 is specified.

Implementation

Folio call implemented in audio folio V30.

Associated Files

<:audio:audio.h>, System.m2/Modules/audio

See Also

ConvertGenericToAudioSignal(), GetAudioSignalInfo(), Instrument

ConvertGenericToAudioSignal

Converts generic signal value to specified audio signal type.

Synopsis

```
float32 ConvertGenericToAudioSignal (int32 signalType, int32
rateDivide,
                                     float32 genericValue)
```

Description

This function converts a generic signal value (e.g., signed floating point value in the range of -1.0 to 1.0) to a signal value of the specified audio signal type (e.g., oscillator frequency in Hertz).

Certain signal types are affected by the execution rate division of the associated instrument:

- AF_SIGNAL_TYPE_SAMPLE_RATE
- AF_SIGNAL_TYPE_OSC_FREQ
- AF_SIGNAL_TYPE_LFO_FREQ

(see description of the Instrument AF_TAG_CALC RATE_DIVIDE tag)

This function does no boundary checking or signal value clipping.

Arguments

signalType

The signal type to convert the generic value to.

rateDivide

The execution rate denominator for the signal (e.g., 1=full rate, 2=half rate, etc).

genericValue

Generic value in the range of -1.0 to 1.0 (for signed signals) or 0 to 2.0 for unsigned signals) to convert.

Return Value

Audio signal value in the units appropriate for signalType. Returns 0 when an undefined signalType or rateDivide of 0 is specified.

Implementation

Folio call implemented in audio folio V30.

Associated Files

<:audio:audio.h>, System.m2/Modules/audio

See Also

ConvertAudioSignalToGeneric(), GetAudioSignalInfo(), Instrument

GetAudioSignalInfo

Get information about audio signal types.

Synopsis

```
Err GetAudioSignalInfo (AudioSignalInfo *info, uint32 infoSize,  
                        uint32 signalType)
```

Description

This function returns range and precision information for a specified audio signal type. The information is returned in an AudioSignalInfo structure.

The AudioSignalInfo structure contains the following fields:

sinfo_Min

Minimum legal value for the signal type.

sinfo_Max

Maximum legal value for the signal type.

sinfo_Precision

The finest amount by which a signal of this signal type can be changed.

Arguments

info

A pointer to an AudioSignalInfo structure where the information will be stored.

infoSize

The size in bytes of the AudioSignalInfo structure.

signalType

The signal type to query. This must be one of the AF_SIGNAL_TYPE_ values defined in <:audio:audio.h>.

Return Value

Non-negative if successful or an error code (a negative value) if an error occurs. On failure, the contents of info are left unchanged.

Implementation

Folio call implemented in audio folio V30.

Caveats

Frequency amounts returned by this function are for instruments running at full execution rate. Divide these values by the AF_TAG_CALCRATE_DIVIDE parameter to compute the correct value for instruments running at other execution rates. This applies to AF_SIGNAL_TYPE_OSC_FREQ, AF_SIGNAL_TYPE_LFO_FREQ, and AF_SIGNAL_TYPE_SAMPLE_RATE.

A simple way to do execution rate conversion is:

```
ConvertGenericToAudioSignal (<signal type>, <rate divide>,  
ConvertAudioSignalToGeneric (<signal type>, 1, <value>));
```

Associated Files

<:audio:audio.h>, System.m2/Modules/audio

See Also

```
ConvertAudioSignalToGeneric(), ConvertGenericToAudioSignal(),  
GetInstrumentPortInfoByIndex(), GetInstrumentPortInfoByName()
```

AbortTimerCue

Cancels a timer request enqueued with `SignalAtAudioTime()`.

Synopsis

```
Err AbortTimerCue (Item Cue)
```

Description

Cancels a timer request enqueued with `SignalAtAudioTime()`. The signal associated with the cue will not be sent if the request is aborted before completion. Aborting a timer request after completion does nothing harmful. Aborting a cue that has not been posted as a timer request is harmless.

Arguments

Cue

Cue item to abort. The task calling this function must own the Cue. The Cue may have already completed - nothing bad will happen.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V21.

Associated Files

<:audio:audio.h>

See Also

`SignalAtAudioTime()`

AudioTimeLaterThan

Compare two AudioTime values with wraparound.

Synopsis

```
int AudioTimeLaterThan (AudioTime t1, AudioTime t2)
```

Description

AudioTimeLaterThan() compares two AudioTime values, taking into account wraparound.

The two time values are assumed to be within 0x7fffffff ticks of each other. The time differences larger than this number will produce incorrect comparisons.

Arguments

t1
First AudioTime value to compare.

t2
Second AudioTime value to compare.

Return Value

TRUE
If t1 is later than t2.

FALSE
If t1 is earlier than or equal to t2.

Implementation

Macro implemented in <:audio:audio.h> V22.

Associated Files

<:audio:audio.h>

See Also

AudioTimeLaterThanOrEqual(), CompareAudioTimes()

AudioTimeLaterThanOrEqual

Compare two AudioTime values with wraparound.

Synopsis

```
int AudioTimeLaterThanOrEqual (AudioTime t1, AudioTime t2)
```

Description

AudioTimeLaterThanOrEqual() compares two AudioTime values, taking into account wraparound.

The two time values are assumed to be within 0x7fffffff ticks of each other. Time differences larger than that will produce incorrect comparisons.

Arguments

t1
First AudioTime value to compare.

t2
Second AudioTime value to compare.

Return Value

TRUE
If t1 is later than or equal to t2.

FALSE
If t1 is earlier than t2.

Implementation

Macro implemented in <:audio:audio.h> V22.

Associated Files

<:audio:audio.h>

See Also

AudioTimeLaterThan(), CompareAudioTimes()

CompareAudioTimes

Compare two AudioTime values with wraparound.

Synopsis

```
int32 CompareAudioTimes (AudioTime t1, AudioTime t2)
```

Description

CompareAudioTimes() compares two AudioTime values taking into account wraparound. The two time values are assumed to be within 0x7fffffff ticks of each other. Time differences larger than this value will produce incorrect comparisons.

Arguments

t1
First AudioTime value to compare.

t2
Second AudioTime value to compare.

Return Value

> 0
If t1 is later than t2.

0
If t1 equals t2.

< 0
If t1 is earlier than t2.

Implementation

Macro implemented in <:audio:audio.h> V22.

Associated Files

<:audio:audio.h>

See Also

AudioTimeLaterThan(), AudioTimeLaterThanOrEqual()

CreateAudioClock

Creates an AudioClock.

Synopsis

```
Item CreateAudioClock (const TagArg *tagList)
```

```
Item CreateAudioClockVA (uint32 tag1, ...)
```

Description

Create a custom audio clock which can run at a different rate than the global audio clock. Custom clocks are useful if you need to change the tempo of MIDI scores.

Call DeleteAudioClock() to dispose of a clock item.

Tag Arguments

None

Return Value

The procedure returns the item number of the clock (a positive value) or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in the `<:audio:audio.h>` V29.

Notes

This function is equivalent to:

```
CreateItem (MKNODEID(AUDIONODE,AUDIO_CLOCK_NODE), tagList)
```

Associated Files

`<:audio:audio.h>`

See Also

```
DeleteAudioClock(),      ReadAudioClock(),      SetAudioClockRate(),  
SetAudioClockRate(), AudioClock
```

DeleteAudioClock

Deletes an AudioClock

Synopsis

```
Err DeleteAudioClock (Item AudioClock)
```

Description

This macro deletes an AudioClock and frees its resources. Any pending timer events for this clock will be triggered as if their time had come in order to prevent deadlocks.

Arguments

```
AudioClock  
    The item number of the AudioClock to delete.
```

Return Value

The procedure returns zero if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>` V29.

Associated Files

`<:audio:audio.h>`

See Also

CreateAudioClock()

GetAudioClockDuration

Asks for the duration of an AudioClock tick in frames.

Synopsis

```
int32 GetAudioClockDuration( Item clock )
```

Description

This procedure asks for the current duration of an AudioClock tick measured in DSP sample frames. The DSP typically runs at 44,100 Hz.

Arguments

```
- clock
    AudioClock item to query, or AF_GLOBAL_CLOCK to query
    system-wide clock.
```

Return Value

Current audio clock duration expressed as the actual number of sample frames between clock interrupts, or a negative error.

Implementation

- Folio call implemented in audio folio V29.

Associated Files

<:audio:audio.h>

See Also

SignalAtAudioTime(), GetAudioClockRate(), SetAudioClockDuration(),
GetAudioFolioInfo()

GetAudioClockRate

Asks for AudioClock rate in Hertz.

Synopsis

```
Err GetAudioClockRate( Item clock, float32 *hertz )
```

Description

This procedure gets an AudioClock rate in Hertz.

Arguments

clock
AudioClock item to query, or AF_GLOBAL_CLOCK to query system-wide clock.

hertz
Pointer to float32 variable to store resulting clock rate in Hertz.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Writes the clock rate to *hertz if successful. Writes nothing if this function returns an error.

Implementation

Folio call implemented in audio folio V29.

Associated Files

<:audio:audio.h>

See Also

SignalAtAudioTime(), GetAudioClockDuration(), SetAudioClockRate()

GetAudioDuration

Asks for the duration of an AF_GLOBAL_CLOCK tick in frames.

Synopsis

```
int32 GetAudioDuration (void)
```

Description

This is a convenience macro that calls GetAudioClockDuration() for AF_GLOBAL_CLOCK.

Return Value

- Current audio clock duration expressed as the actual number of sample frames between clock interrupts, or a negative error.

Implementation

Macro implemented in `<:audio:audio.h>` V29.

Associated Files

`<:audio:audio.h>`

- See Also

GetAudioClockDuration()

GetAudioTime

Reads AF_GLOBAL_CLOCK time.

Synopsis

```
AudioTime GetAudioTime (void)
```

Description

This procedure returns the current 32-bit audio time from AF_GLOBAL_CLOCK. This timer is derived from the Audio DAC's 44100 Hz clock so it can be used to synchronize events with an audio stream. This timer typically runs at approximately, but not exactly, 240 Hz. Use GetAudioDuration() for an exact correlation between sample frames and audio timer ticks.

Return Value

The procedure returns the current audio time expressed in audio ticks.

Implementation

Convenience call implemented in audio folio V20.

Notes

This function is similar to:

```
ReadAudioClock (AF_GLOBAL_CLOCK, &time)
```

Associated Files

<:audio:audio.h>

See Also

```
SignalAtAudioTime(), GetAudioClockRate(), GetAudioClockDuration(),  
GetAudioFolioInfo(), ReadAudioClock()
```

ReadAudioClock

Reads AudioTime from an AudioClock.

Synopsis

```
Err ReadAudioClock ( Item clock, AudioTime *timePtr )
```

Description

This procedure returns the current 32-bit audio time. This timer is derived from the Audio DAC's 44100 Hz clock so it can be used to synchronize events with an audio stream. This timer typically runs at approximately, but not exactly, 240 Hz. Use GetAudioClockDuration() for an exact correlation between sample frames and audio timer ticks. — — —

Arguments

clock

Item number of an AudioClock or AF_GLOBAL_CLOCK.

timePtr

Pointer to AudioTime variable that will be set to the current time.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Writes AudioTime to *timePtr if successful. Writes nothing to *timePtr this function fails.

Implementation

Folio call implemented in audio folio V29.

Associated Files

<:audio:audio.h>

See Also

SignalAtAudioTime(), GetAudioClockRate(), GetAudioClockDuration(), GetAudioFolioInfo(), GetAudioTime()

SetAudioClockDuration

Changes duration of AudioClock tick.

Synopsis

```
Err SetAudioClockDuration (Item clock, int32 numFrames)
```

Description

This procedure changes the rate of an audio clock by specifying a duration for the clock's tick. The clock is driven by a countdown timer in the DSP. When the DSP timer reaches zero, it increments the audio clock by one. The DSP timer is decremented at the sample rate of 44,100 Hz. Thus tick duration is given in units of sample frames.

The current audio clock duration can be read with `GetAudioClockDuration()`.

Arguments

`clock`
Item number of an AudioClock.

`numFrames`
Duration of one audio clock tick in sample frames at 44,100 Hz.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V29.

Associated Files

`<:audio:audio.h>`

See Also

`GetAudioClockDuration()`, `SetAudioClockRate()`, `GetAudioClockRate()`, `SignalAtAudioTime()`, `SleepUntilAudioTime()`, `GetAudioFolioInfo()`

SetAudioClockRate

Changes the rate of the AudioClock.

Synopsis

```
Err SetAudioClockRate (Item clock, float32 hertz)
```

Description

This procedure changes the rate of the audio clock. The default rate is 240 Hz. The range of the clock rate is 60 to system sample rate.

The current audio clock rate can be read with `GetAudioClockRate()`.

Arguments

clock

Item number of an AudioClock.

hertz

Rate value in Hz. Must be in the range of 60 Hz to system sample rate.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V29.

Associated Files

<:audio:audio.h>

See Also

`GetAudioClockRate()`, `SetAudioClockDuration()`,
`GetAudioClockDuration()`, `SignalAtAudioTime()`, `SleepUntilAudioTime()`

SignalAtAudioTime

Requests a wake-up call at a given time.

Synopsis

```
Err SignalAtAudioTime ( Item clock, Item cue, AudioTime time)
```

Description

This procedure requests that a signal be sent to the calling item at the specified audio time. The cue item can be created using `CreateCue()`.

A task can get the cue's signal mask using `GetCueSignal()`. The task can then call `WaitSignal()` to enter wait state until the cue's signal is sent at the specified time.

If you need to, you can call `AbortTimerCue()` to cancel the timer request before completion.

See the example program `ta_timer` for a complete example.

Arguments

`clock`
Item number of an `AudioClock` or `AF_GLOBAL_CLOCK`.

`cue`
Item number of a cue.

`time`
The time at which to send a signal to the calling task.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in `audio folio V29`.

Associated Files

`<:audio:audio.h>`

See Also

`AbortTimerCue()`, `CreateCue()`, `GetCueSignal()`, `SleepUntilAudioTime()`, `GetAudioClockRate()`

SignalAtTime

Requests a wake-up call at a given time from AF_GLOBAL_CLOCK.

Synopsis

```
Err SignalAtTime (Item cue, AudioTime time)
```

Description

Convenience macro that calls SignalAtAudioTime() with AF_GLOBAL_CLOCK.

Arguments

cue

Item number of a cue.

time

The time at which to send a signal to the calling task.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:audio.h>` V29.

Notes

This macro is equivalent to:

```
SignalAtAudioTime (AF_GLOBAL_CLOCK, cue, time)
```

Associated Files

`<:audio:audio.h>`

See Also

AbortTimerCue(), CreateCue(), GetCueSignal(), SleepUntilTime(),
GetAudioClockRate(), SignalAtAudioTime()

SleepUntilAudioTime

Enters wait state until AudioClock reaches a specified AudioTime.

Synopsis

```
Err SleepUntilAudioTime (Item clock, Item cue, AudioTime time)
```

Description

This procedure will not return until the specified audio time is reached. If that time has already passed, it will return immediately. If it needs to wait, your task will enter wait state so that it does not consume CPU time.

Arguments

clock
Item number of an AudioClock or AF_GLOBAL_CLOCK.

cue
The item number of the cue used to wait. This Cue shouldn't be used for anything else, or this function might return early.

time
AudioTime to wait for.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V29.

Examples

```
// If you want to wait for a specific number of ticks to go by,  
// you can get the current audio time from ReadAudioClock() and  
// add your delay:
```

```
AudioTime time;  
  
ReadAudioClock (clock, &time);  
SleepUntilAudioTime (clock, cue, time + DELAYTICKS);
```

Notes

This function is equivalent to (shown with error handling removed):

```
SignalAtAudioTime (clock, cue, time);  
WaitSignal (GetCueSignal(cue));
```

Associated Files

<:audio:audio.h>

See Also

SignalAtAudioTime(), CreateCue(), GetCueSignal(), GetAudioTime(),
GetAudioClockRate()

SleepUntilTime

Enters wait state until
AF_GLOBAL_CLOCK reaches a specified
AudioTime.

Synopsis

```
Err SleepUntilTime (Item cue, AudioTime time)
```

Description

Convenience macro that calls SleepUntilAudioTime() for
AF_GLOBAL_CLOCK.

Arguments

cue
The item number of the cue used to wait. This Cue shouldn't
be used for anything else, or this function might return
early.

time
AudioTime to wait for.

Return Value

The procedure returns a non-negative value if successful or an error
code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:audio.h> V29.

Examples

```
// If you want to wait for a specific number of ticks to go by,  
// you can get the current audio time from GetAudioTime() and  
// add your delay:  
SleepUntilTime (cue, GetAudioTime() + DELAYTICKS);
```

Notes

This macro is equivalent to:

```
SleepUntilAudioTime (AF_GLOBAL_CLOCK, cue, time)
```

Associated Files

<:audio:audio.h>

See Also

SleepUntilAudioTime(), GetAudioTime()

ArmTrigger

Assigns a Cue to an Instrument's Trigger.

Synopsis

```
Err ArmTrigger (Item instrument, const char *triggerName, Item cue,
               uint32 flags)
```

Description

Assigns a Cue to be sent the next time that an Instrument's Trigger goes off.

Triggers can be set to either continuous or one-shot mode by this function. In one-shot mode, the Trigger is automatically disarmed after the assigned Cue is sent (as if DisarmTrigger() had been called). In this mode, the Trigger must be re-armed by another call to ArmTrigger() in order to have the Cue sent for a subsequent Trigger event.

In continuous mode, the assigned Cue is sent every time the Trigger goes off. It is never automatically disarmed; there's no need to manually re-arm it after receiving the Cue.

A Trigger is normally only reset when its associated Cue is sent. This means if you use a one-shot Trigger, you won't lose Trigger events that happen while the Trigger is disarmed. If a Trigger event has already occurred by the time the Trigger is armed, the Cue is sent immediately upon arming the Trigger. You can manually reset the Trigger when you arm it if you want to have the Cue sent the `_next_` time the Trigger goes off, and not for any Trigger events that may have occurred while the Trigger was disarmed.

A Trigger can be armed, re-armed, or disarmed at any time. The Trigger mode can be changed between one-shot and continuous at any time. The most recent call to ArmTrigger() takes precedence for any given Trigger. Use DisarmTrigger() (a convenience macro) to disarm the trigger prior to disposing of the Cue assigned to the Trigger.

The audio folio automatically disarms the Trigger when either the owning Instrument or assigned Cue are deleted while the Trigger is armed.

Arguments

instrument

Instrument Item containing trigger to arm.

triggerName

Name of Instrument's Trigger. A NULL pointer causes the default name "Trigger" to be used.

cue

Cue Item to assign to Trigger.

flags

Optional set of flags described below, which may be used in any combination.

AF_F_TRIGGER_CONTINUOUS

When set, causes Trigger to be armed in continuous mode. Otherwise, Trigger is armed in one-shot mode.

AF_F_TRIGGER_RESET

Causes Trigger to be reset before being armed. Any accumulated Trigger events that have occurred before this call is flushed. When not set, any previously occurring event from this Trigger would cause the Cue to be sent immediately.

Return Value

A non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V27.

Caveats

Care should be used when using AF_F_TRIGGER_CONTINUOUS mode. Use of this mode causes the audio folio's high priority task to re-enable a DSP software interrupt immediately after it has been serviced. If this interrupt occurs very frequently, the only task that could get time to run are the audio folio's task and anything of equal or higher priority (i.e., system code only).

A good example of what NOT to do is connect schmidt_trigger.dsp to the output of an oscillator running at a really high frequency followed by arming schmidt_trigger.dsp's trigger in AF_F_TRIGGER_CONTINUOUS mode.

Associated Files

<:audio:audio.h>

See Also

DisarmTrigger(), schmidt_trigger.dsp, CreateCue(),
MonitorAttachment()

DisarmTrigger

Remove previously assigned Cue from an Instrument's Trigger.

Synopsis

```
Err DisarmTrigger (Item instrument, const char *triggerName)
```

Description

Disarms a Trigger previously armed by ArmTrigger(). Call this function to clean up in an orderly fashion before deleting the Cue that was last given to ArmTrigger(). Redundant, but harmless, if Trigger is already disarmed (either by a previous call to DisarmTrigger() or if the Trigger is in one-shot mode and has already gone off).

Arguments

instrument

Instrument Item containing trigger to disarm.

triggerName

Name of Instrument's Trigger. A NULL pointer causes the default name "Trigger" to be used.

Return Value

A non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:audio.h> V27.

Associated Files

<:audio:audio.h>

See Also

ArmTrigger()

Convert12TET_FP

Converts a pitch bend value in semitones and cents into a float value.

Synopsis

```
Err Convert12TET_FP (int32 semitones, int32 cents, float32
*fractionPtr)
```

Description

This procedure, whose name reads (convert 12-tone equal-tempered to float) converts a pitch bend value expressed in semitones and cents to a float bend value. The bend value is used with `BendInstrumentPitch()` to bend the instrument's pitch up or down: the value multiplies the frequency of the instrument's output to create a resultant pitch bent up or down by the specified amount.

Note that the semitones and cents values need not both be positive or both be negative. One value can be positive while the other value is negative. For example, -5 semitones and +30 cents bends pitch down by 4 semitones and 70 cents.

`Convert12TET_FP()` stores the resultant bend value in `fractionPtr`.

Arguments

`semitones`

The integer number of semitones to bend up or down (can be negative or positive).

`cents`

The integer number of cents (from -100 to 100) to bend up or down.

`fractionPtr`

Pointer to a float32 variable where the bend value will be stored.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

`BendInstrumentPitch()`

CreateTuning

Creates a Tuning Item.

Synopsis

```
Item CreateTuning (const float32 *frequencies, int32 numNotes,  
                  int32 notesPerOctave, int32 baseNote)
```

Description

This procedure creates a tuning item that can be used to tune an instrument.

NotesPerOctave is set to 12 for a standard western tuning system. For every octave's number of notes up or down, the frequency will be doubled or halved. You can thus specify 12 notes of a 12-tone scale, then extrapolate up or down for the other octaves. You should specify the entire range of pitches if the tuning does not repeat itself every octave. For more information, see the Music Programmer's Guide.

When you are finished with the tuning item, you should call DeleteTuning() to deallocate the resources.

Arguments

frequencies

A pointer to an array of float32 frequencies, which lists a tuning pattern.

numNotes

The number of frequencies in the array. This value must be \geq notesPerOctave.

notesPerOctave

The number of notes in an octave (12 in a standard tuning scale).

baseNote

The MIDI pitch (note) value of the first frequency in the array. If your array starts with the frequency of middle C, its MIDI pitch value would equal 60.

Return Value

The procedure returns the item number for the tuning or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in libc.a V29.

Notes

This function is equivalent to:

```
CreateItemVA (MKNODEID(AUDIONODE,AUDIO_TUNING_NODE),  
              AF_TAG_ADDRESS,      frequencies,  
              AF_TAG_FRAMES,      numIntervals,
```

```
AF_TAG_NOTESPEROCTAVE, notesPerOctave,  
AF_TAG_BASENOTE,      baseNote,  
TAG_END);
```

Associated Files

<:audio:audio.h>, libc.a

See Also

Tuning, DeleteTuning(), TuneInsTemplate(), TuneInstrument()

DeleteTuning

Deletes a Tuning.

Synopsis

```
Err DeleteTuning (Item tuning)
```

Description

This procedure deletes the specified Tuning and frees any resources dedicated to it. Note that if you delete a tuning that's in use for an existing instrument, that instrument reverts to its default tuning.

If the Tuning was created with { AF_TAG_AUTO_FREE_DATA, TRUE }, then the tuning data is also be freed when the Tuning Item is deleted. Otherwise, the tuning data is not freed automatically.

Arguments

tuning
The item number of the Tuning to delete.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:audio.h> V27.

Associated Files

<:audio:audio.h>

See Also

Tuning, CreateTuning()

TuneInsTemplate

Applies the specified Tuning to an instrument Template.

Synopsis

```
Err TuneInsTemplate (Item insTemplate, Item tuning)
```

Description

This procedure applies the specified tuning to the specified instrument template. When an instrument is created using the template, the tuning is applied to the new instrument. When notes are played on the instrument, they play using the specified tuning system.

Arguments

insTemplate
The item number of the template.

tuning
The item number of the tuning.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

CreateTuning(), TuneInstrument()

TuneInstrument

Applies the specified Tuning to an Instrument.

Synopsis

```
Err TuneInstrument (Item instrument, Item tuning)
```

Description

This procedure applies the specified Tuning to an instrument so that notes played on the instrument use that tuning. If no tuning is specified, the default 12-tone equal-tempered tuning is used.

Arguments

instrument

The item number of the instrument.

tuning

The item number of the tuning.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in audio folio V20.

Associated Files

<:audio:audio.h>, libc.a

See Also

CreateTuning(), TuneInsTemplate()

Chapter 2

Audio Patch File Folio Calls

This section presents the reference documentation for the AudioPatchFile folio, which provides services to load patch (FORM 3PCH) files. These files are created by tools such as makepatch and ARIA.

--Patch-File-Overview--

Overview of the FORM 3PCH binary patch file format.

File Format

FORM 3PCH {

Patch Compiler Input - This FORM represents the PatchCmd list to be given to the patch compiler. There must be exactly 1 FORM PCMD in the FORM 3PCH (not counting nesting).

FORM PCMD {

Constituent Template list for Patch. Any number and order of these chunks/FORMs. The resulting templates are referenced by index by the PCMD chunk.

```
PTMP {
    PackedStringArray    // packed string array of .dsp
    template names
}
PMIX {
    MixerSpec[]          // array of MixerSpecs
}
FORM 3PCH {              // nested patch
    .
    .
    .
}
```

Packed string array of all strings required by PatchCmd list.

Indexed by keys stored in PCMD chunk. There must be one of these in the FORM PCMD.

```
PNAM {
    PackedStrings
}
```

Array of file-conditioned PatchCmds terminated by a PATCH_CMD_END:

- char * are stored as uint32 byte offset into PNAM chunk + 1, NULL is 0.

- InsTemplate items are stored as index into List of templates defined

stored in PTMP, PMIX, and nested FORM 3PCHs in order of appearance in the FORM PCMD.

```
PCMD {
    PatchCmd[]
}
```

Tuning - Defines a custom tuning for the the Patch Template.
Either 0
or 1 of these FORMs (*** not yet supported)

```
FORM PTUN {
    ATAG {
        AudioTagHeader AUDIO_TUNING_NODE
    }
    BODY {
        float32 TuningData[]
    }
}
```

Attachments - Each FORM PATT defines a slave Item and attachments for that Item. There may be any number of these FORMs in a FORM 3PCH. If present, they must appear after the FORM PCMD.

Each PATT chunk within a FORM PATT defines one attachment between the slave defined by the FORM PATT and the patch. There must be at least one PATT chunk in each FORM PATT.

```
FORM PATT {
    PATT {
        PatchAttachment
    }
    PATT {
        PatchAttachment
    }
    .
    .
    .
    ATAG {
        AudioTagHeader
    }
    BODY {
        data
    }
}
```

Description

This is an IFF FORM which is capable of storing a PatchCmd list and list of attachments for a Patch Template. The function LoadPatchTemplate() loads this file format. The shell program makepatch and the graphical patch creation program ARIA write patch files in this format.

The AudioPatchFile folio offers two ways to parse a FORM 3PCH:

LoadPatchTemplate()

Loads a FORM 3PCH out of the named file. This is the simplest way to load a patch written by makepatch or ARIA.

EnterForm3PCH(), ExitForm3PCH()

These two functions are IFF Folio entry and exit-handlers for FORM 3PCH. You may install these handlers, your own FORM 3PCH handlers which then call these, or a combination. Access to the handlers permits you to parse one or more FORM 3PCHs embedded in a larger IFF context (e.g., a score file).

You don't need to bother with these functions if all you want to do is load a patch made by makepatch or ARIA. Just use LoadPatchTemplate(), which is a client of these handlers.

In order to facilitate simple and complete deletion of a loaded patch template, the parser creates slave items (for Samples and Envelopes loaded from the patch file) with { AF_TAG_AUTO_FREE_DATA, TRUE }, and Attachments to these slave items with { AF_TAG_AUTO_DELETE_SLAVE, TRUE }.

Caveats

Delay lines, like other attached items, belong to the patch template, so all instruments created from such a patch share the same delay line. This fact cripples the delay line feature for use in any application requiring the creation of multiple instruments from a patch. This feature may still be used in cases where only one instrument is to be allocated from the patch.

If you need a delay line per instrument, create the patch with the delay related instruments but without the delay line itself. When you create instruments from the patch, create and attach the delay line yourself. You may make use of the attachment tag AF_TAG_AUTO_DELETE_SLAVE to simplify cleanup of the instrument delay lines.

Doesn't yet support custom tuning information stored in patches (FORM PTUN).

Attachments in nested patches aren't propagated to the enclosing patch (a limitation of the patch compiler).

Associated Files

<:audio:patchfile.h>, <:audio:patch.h>, <:audio:atag.h>, System.m2/Modules/audiopatch, System.m2/Modules/audiopatchfile

See Also

LoadPatchTemplate(), EnterForm3PCH(), ExitForm3PCH(), PatchCmd, makepatch, Template, Sample, Envelope, Tuning, Attachment

EnterForm3PCH

Standard FORM 3PCH entry handler.

Synopsis

```
Err EnterForm3PCH (IFFParser *iff, const char *patchName)
```

Description

This function creates context data and registers additional chunks and handlers for parsing a FORM 3PCH patch. Either install this function as the FORM 3PCH entry handler, or call it from your own FORM 3PCH entry handler.

This function must be used in conjunction with `ExitForm3PCH()`.

Arguments

`iff`

IFFParser positioned at the entry of a FORM 3PCH.

`patchName`

Name to give the Patch Template Item created and returned by `ExitForm3PCH()`, or NULL to leave the Item unnamed. This string is copied when the Item is created, but not until then. So the string must remain valid until the FORM 3PCH context is exited.

Return Value

The procedure returns `IFF_CB_CONTINUE` if successful, or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in `AudioPatchFile Folio V30`.

Module Open Requirements

`OpenAudioFolio()`, `OpenAudioPatchFileFolio()`, `OpenIFFFolio()`

Associated Files

`<:audio:patchfile.h>`, `System.m2/Modules/audio`, `System.m2/Modules/audiopatchfile`,
`System.m2/Modules/iff`

See Also

`--Patch-File-Overview--`, `ExitForm3PCH()`, `InstallEntryHandler()`,
`LoadPatchTemplate()`

ExitForm3PCH

Standard FORM 3PCH exit handler.

Synopsis

```
Item ExitForm3PCH (IFFParser *iff, void *dummy)
```

Description

This function completes the FORM 3PCH parsing started by `EnterForm3PCH()`, and returns the Item number of the completed Patch Template. Either install this function as the FORM 3PCH exit handler, or call it from your own FORM 3PCH exit handler.

Because this function returns an Item number instead of `IFF_CB_CONTINUE`, when used as the exit handler (instead of being called by your own) it causes `ParseIFF()` to stop parsing the IFF stream and to return this item number. `LoadPatchTemplate()` takes advantage of this behavior.

This function must be used in conjunction with `EnterForm3PCH()`.

Arguments

`iff`

IFFParser positioned at the exit of a FORM 3PCH.

`dummy`

This argument is isn't used, and is here only to fit the `IFFCallback` prototype. Set it to `NULL`.

Return Value

The procedure returns a Patch Template Item number (a non-negative value) if successful, or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in `AudioPatchFile Folio V30`.

Examples

Assume that we are collecting a list of patches from an IFF file (error handling omitted). Use a custom FORM 3PCH handler to put the results of `ExitForm3PCH()` into our list.

```
{
    .
    .
    if ((errcode = InstallEntryHandler (iff, ID_3PCH, ID_FORM,
    IFF_CIL_TOP,
        (IFFCallback)EnterForm3PCH, NULL)) < 0) goto clean;
    if ((errcode = InstallExitHandler (iff, ID_3PCH, ID_FORM,
    IFF_CIL_TOP,
        (IFFCallback)MyExitForm3PCH, &patchList)) < 0) goto clean;
    .
    .
    if ((errcode = ParseIFF (iff, IFF_PARSE_SCAN)) < 0) goto clean;
    .
    .
}
```

```

}

// custom exit handler - gets Patch Template Item from FORM 3PCH
parser
// and adds to to patchList, calling our
Err MyExitForm3PCH (IFFParser *iff, List *patchList)
{
    Err errcode;
    Item patchTemplate;

    // get patch template from FORM 3PCH handler by calling
    ExitForm3PCH()
    if ((errcode = patchTemplate = ExitForm3PCH (iff, NULL))) < 0)
goto clean;

    // add patch to list
    if ((errcode = AddPatchToList (patchList, patchTemplate)) < 0)
goto clean;

    // success: keep parsing
    return IFF_CB_CONTINUE;
}

clean:
    // failure: delete patch (because it didn't get added to
list)
    // and return error code.
    DeleteItem (patchTemplate);
    return errcode;
}

Err AddPatchToList (List *patchList, Item patchTemplate)
{
    .
    .
    .
}

```

Module Open Requirements

OpenAudioFolio(), OpenAudioPatchFileFolio(), OpenIFFFolio()

Associated Files

```
<:audio:patchfile.h>,                                     System.m2/Modules/audio,
System.m2/Modules/audiopatchfile, System.m2/Modules/iff
```

See Also

```
--Patch-File-Overview--      EnterForm3PCH(),      InstallExitHandler(),
LoadPatchTemplate()
```

LoadPatchTemplate

Loads a binary patch file (FORM 3PCH).

Synopsis

```
Item LoadPatchTemplate (const char *fileName)
```

Description

This function loads a binary Patch Template file prepared by audio tools such as makepatch and ARIA, and returns a Patch Template Item. The Patch Template Item is given the same name as the patch file.

Unload it with `UnloadPatchTemplate()`.

This function assumes that the patch you want to load is in a discrete file. To parse a FORM 3PCH in a larger IFF context, use `EnterForm3PCH()` and `ExitForm3PCH()`. See `--Patch-File-Overview--` for more information on this.

Arguments

`fileName`

Name of patch template file to load. This name is also given to the Patch Template Item.

Return Value

The procedure returns an item number of the Template if successful (a non-negative value) or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in `AudioPatchFile Folio V30`.

Notes

This function is equivalent to:

```
Item LoadPatchTemplate (const char *fileName)
{
    IFFParser *iff;
    Item result;

    // Open IFF file (sets iff to NULL on failure).
    if ((result = CreateIFFParserVA (&iff, FALSE,
        IFF_TAG_FILE, fileName,
        TAG_END)) < 0) goto clean;

    // Install FORM 3PCH entry handler.
    // Pass the fileName to it so that the resulting Item is
    named the
    // same as the patch file.
    if ((result = InstallEntryHandler (iff, ID_3PCH, ID_FORM,
        IFF_CIL_TOP,
        (IFFCallback)EnterForm3PCH, fileName)) < 0) goto clean;
```

```
        // Install FORM 3PCH exit handler.
        // Take advantage of the fact that ExitForm3PCH() returns
the      // completed Item number, which causes ParseIFF() to stop
parsing  // and return it.
        if ((result = InstallExitHandler (iff, ID_3PCH, ID_FORM,
IFF_CIL_TOP,
        (IFFCallback)ExitForm3PCH, NULL)) < 0) goto clean;

        // Parse file: returns completed Patch Template Item on
success  // (because of what ExitForm3PCH() returns).
        // Translate otherwise meaningless IFF completion codes into
suitable // errors.
        switch (result = ParseIFF (iff, IFF_PARSE_SCAN)) {
            case IFF_PARSE_EOF:
                result = PATCHFILE_ERR_MANGLED;
                break;
        }

clean:
    DeleteIFFParser (iff);
    return result;
}
```

Caveats

Doesn't yet support custom tuning information stored in patches (FORM PTUN).

Attachments in nested patches aren't propagated to the enclosing patch (a limitation of the patch compiler).

Module Open Requirements

OpenAudioFolio(), OpenAudioPatchFileFolio()

Associated Files

<:audio:patchfile.h>, System.m2/Modules/audio,
System.m2/Modules/audiopatchfile

See Also

--Patch-File-Overview--, UnloadPatchTemplate(), LoadScoreTemplate(),
EnterForm3PCH(), ExitForm3PCH(), CreatePatchTemplate()

UnloadPatchTemplate

Unloads a Patch Template loaded by LoadPatchTemplate().

Synopsis

```
Err UnloadPatchTemplate (Item patchTemplate)
```

Description

Unloads a patch template loaded by LoadPatchTemplate(). This has the same side effects as UnloadInsTemplate() (e.g., deleting instruments, attachments, slaves of AF_TAG_AUTO_DELETE_SLAVE attachments, etc.)

Arguments

patchTemplate
Patch Template Item to unload.

Return Value

The procedure returns a non-negative value on success or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:patchfile.h> V29.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:patchfile.h>, System.m2/Modules/audio

See Also

LoadPatchTemplate()

Chapter 3

Audio Patch Folio Calls

This section presents the reference documentation for the AudioPatch folio, which provides services to create custom instrument templates.

CreatePatchTemplate

Constructs a custom Patch Template from simple Instrument Templates.

Synopsis

```
Item CreatePatchTemplate (const PatchCmd *patchCmdList, const TagArg *tagList)
```

```
Item CreatePatchTemplateVA (const PatchCmd *patchCmdList, uint32 tag1, ...)
```

Description

This function creates a Patch Template from a list of PatchCmds. Patches may consist of zero or more Instrument Templates, zero or more defined ports and knobs, and optional internal connections between these.

The PatchCmd parser makes several passes through the PatchCmd list, one per class defined below. Within each class, PatchCmds are acted upon in order of appearance. The PatchCmd class order is:

1. PATCH_CMD_SET_COHERENCE
2. PATCH_CMD_ADD_TEMPLATE
3. PATCH_CMD_DEFINE_PORT and PATCH_CMD_DEFINE_KNOB
4. PATCH_CMD_EXPOSE
5. PATCH_CMD_CONNECT and PATCH_CMD_SET_CONSTANT

Because of the class-based parsing, some order independence is offered in constructing a PatchCmd list. For example, it makes no difference whether things that are to be connected (PATCH_CMD_ADD_TEMPLATE, PATCH_CMD_DEFINE_PORT, or PATCH_CMD_DEFINE_KNOB) are defined before a connection (PATCH_CMD_CONNECT) is defined.

The resource requirements for the compiled patch are typically a little less than the sum of the resource requirements of the constituent templates plus data memory for the patch's outputs and knobs (one word each). If PATCH_CMD_SET_COHERENCE is set to FALSE, the instrument code is packed together, thus shrinking the code by a couple of words for each constituent instrument.

Call DeletePatchTemplate() when done with this Template.

Arguments

patchCmdList

Pointer to a PatchCmd list from which to create a patch.

Tag Arguments

TAG_ITEM_NAME (const char *)

Optional name for the newly created patch. Defaults to not having a name.

Return Value

The procedure returns an instrument Template Item number (a non-negative value) if successful or an error code (a negative value) if an error occurs.

Implementation

Folio call implemented in AudioPatch Folio V30.

Caveats

Items attached to any of the constituent Template Items of a patch (i.e. Samples, Envelopes, or Tunings), are not propagated to the Patch Template. If you wish to do this, you must Attach these things to the compiled Patch Template yourself. This is a bit of a gotcha when it comes to including previously defined Patch Templates in another Patch, because it requires you to know what, if anything, is attached to this Patch Template. (there are audio folio services to help you identify these however: see the documentation for GetAttachments() and the Attachment tags)

Module Open Requirements

OpenAudioFolio(), OpenAudioPatchFolio()

Associated Files

<:audio:patch.h>, System.m2/Modules/audio, System.m2/Modules/audiopatch

See Also

PatchCmd, CreatePatchCmdBuilder(), Template, DeletePatchTemplate(), LoadInsTemplate(), CreateMixerTemplate(), CreateInstrument(), LoadPatchTemplate(), makepatch

DeletePatchTemplate

Deletes a custom Instrument Template created by CreatePatchTemplate()

Synopsis

```
Err DeletePatchTemplate (Item patchTemplate)
```

Description

This function deletes a Patch Template created by CreatePatchTemplate(). This has the same side effects as UnloadInsTemplate() (e.g., deleting instruments, attachments, slaves of AF_TAG_AUTO_DELETE_SLAVE attachments, etc.)

Arguments

```
patchTemplate  
    Patch Template Item to delete.
```

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in <:audio:patch.h> V27.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

```
<:audio:patch.h>, System.m2/Modules/audio
```

See Also

```
CreatePatchTemplate(), PatchCmd
```

DumpPatchCmd

Prints out a PatchCmd.

Synopsis

```
void DumpPatchCmd (const PatchCmd *patchCmd, const char *prefix)
```

Description

Prints out the contents of a PatchCmd.

Arguments

patchCmd
PatchCmd to print.

prefix
Text to print on line before PatchCmd. Can be NULL.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, DumpPatchCmdList()

DumpPatchCmdList

Prints out a PatchCmd list.

Synopsis

```
void DumpPatchCmdList (const PatchCmd *patchCmdList, const char
*banner)
```

Description

Prints out the contents of a PatchCmd list.

Arguments

patchCmdList

The list of PatchCmds to print. Can be NULL, which signifies an empty list.

banner

Description of PatchCmd list to print. Can be NULL.

Implementation

Library call implemented in libc.a V29.

Module Open Requirements

OpenAudioPatchFolio()

Associated Files

<:audio:patch.h>, libc.a, System.m2/Modules/audiopatch

See Also

PatchCmd, DumpPatchCmd()

NextPatchCmd

Finds the next PatchCmd in a PatchCmd list.

Synopsis

```
PatchCmd *NextPatchCmd (const PatchCmd **cmdState)
```

Description

This function iterates through a PatchCmd list, skipping and chaining as dictated by control commands. There are three control commands:

PATCH_CMD_NOP

Ignores that single entry and moves to the next one.

PATCH_CMD_JUMP

Has a pointer to another array of PatchCmds.

PATCH_CMD_END

Marks the end of the PatchCmd list.

This function only returns PatchCmds which are not control commands. Each call returns either the next PatchCmd you should examine, or NULL when the end of the list has been reached.

Arguments

cmdState

This is a pointer to a storage location used by the iterator to keep track of its current location in the PatchCmd list. The variable that this parameter points to should be initialized to point to the first PatchCmd in the list, and should not be changed thereafter.

Return Value

This function returns a pointer to a PatchCmd structure, or NULL if all the PatchCmds have been visited. None of the control commands are ever returned to you, they are handled transparently by this function.

Implementation

Folio call implemented in AudioPatch Folio V30.

Example

```
void WalkPatchCmdList (const PatchCmd *patchCmdList)
{
    const PatchCmd *state, *pc;

    for (state = patchCmdList; pc = NextPatchCmd(&state);) {

        // do something with pc

    }
}
```

Module Open Requirements

OpenAudioPatchFolio()

Associated Files

<:audio:patch.h>, System.m2/Modules/audiopatch

See Also

PatchCmd, CreatePatchTemplate(), NextTagArg(), PATCH_CMD_JUMP,
PATCH_CMD_NOP, PATCH_CMD_END

PatchCmd

Command set used by `CreatePatchTemplate()`.

Description

PatchCmds are the command set used by `CreatePatchTemplate()` to construct a custom patch template. A PatchCmd list is an array of PatchCmds terminated by `PATCH_CMD_END` or a list of PatchCmd arrays joined by `PATCH_CMD_JUMP`, and the last array terminated by `PATCH_CMD_END`. In this respect PatchCmd lists are much like TagArg lists.

Unlike TagArgs, each `PATCH_CMD_` has a different set of arguments. So, there is a different structure associated with each `PATCH_CMD_`. In order to make an array out of these different structures, the PatchCmd typedef is defined as a union of all of these structures.

Using a union complicates defining a PatchCmd array with static initializers, however, so there is a simple PatchCmd builder system (see `CreatePatchCmdBuilder()`), which offers a set of functions to construct a PatchCmd list one PatchCmd at a time.

Names used in PatchCmds (e.g., port, block, knob) are all matched case-insensitively and must be unique within their name space. Blocks have their own name space. Ports, knobs, exposed things, and imported symbols (internal to DSP instruments) occupy the same name space.

Associated Files

`<:audio:patch.h>`

See Also

`CreatePatchTemplate()`, `CreatePatchCmdBuilder()`, `NextPatchCmd()`,
`DumpPatchCmdList()`, `NextTagArg()`, `makepatch`

PATCH_CMD_ADD_TEMPLATE Adds an instrument Template to patch.**Synopsis**

```
typedef struct PatchCmdAddTemplate {  
    uint32      pc_CmdID;           // PATCH_CMD_ADD_TEMPLATE  
    const char  *pc_BlockName;  
    Item        pc_InsTemplate;  
    uint32      pc_Pad3;  
    uint32      pc_Pad4;  
    uint32      pc_Pad5;  
    uint32      pc_Pad6;  
} PatchCmdAddTemplate;
```

Description

Add an Instrument Template Item to the patch with the specified name. Any Instrument Template may be added to the patch: standard DSP instruments, patches, or mixers. Multiple instances of the same Instrument Template Item may be added to any patch. Each template must be given a unique block name within the patch.

Fields

pc_BlockName
Block name for template being added. Each block in a patch must have a unique name. Names are compared case-insensitively. Internal connections are specified using this block name.

pc_InsTemplate
Instrument Template Item to use for this block. May be a standard DSP instrument template, a mixer, or a previously created patch.

Notes

To minimize propagation delay across a patch, instruments should be added in the order of signal flow.

Caveats

Any Items attached to pc_InsTemplate (i.e. Samples, Envelopes, or Tunings) are not propagated to the compiled patch.

Associated Files

<:audio:patch.h>

See Also

CreatePatchTemplate(), PatchCmd, AddTemplateToPatch(), PATCH_CMD_EXPOSE, PATCH_CMD_CONNECT, PATCH_CMD_SET_CONSTANT, PATCH_CMD_SET_COHERENCE

PATCH_CMD_CONNECT

Connects patch blocks and ports to one another.

Synopsis

```
typedef struct PatchCmdConnect {  
    uint32      pc_CmdID;           // PATCH_CMD_CONNECT  
    const char *pc_FromBlockName;  
    const char *pc_FromPortName;  
    uint32      pc_FromPartNum;  
    const char *pc_ToBlockName;  
    const char *pc_ToPortName;  
    uint32      pc_ToPartNum;  
} PatchCmdConnect;
```

Description

Creates an internal signal connection between a patch input, patch knob, or block output, and a patch output, block input, or block knob.

Any block input or block knob can have up to one connection or constant. Unused block inputs default to 0; unused block knobs default to their default value. All patch outputs must be connected.

Fields

pc_FromBlockName

Name of a patch block containing an output to connect from. Or NULL to indicate that the source of the connection is a patch input or patch knob. (patch inputs and knobs appear as outputs for the purpose of internal connections)

pc_FromPortName

Name of an output of the block named by pc_FromBlockName, or patch input or patch knob when pc_FromBlockName is NULL.

pc_FromPartNum

Part number of the block output, patch input, or patch knob specified by pc_FromBlockName and pc_FromPortName.

pc_ToBlockName

Name of a patch block containing an input or knob to connect to. Or NULL to indicate that the destination of the connection is a patch output. (patch outputs appear as inputs for the purpose of internal connections)

pc_ToPortName

Name of an input or knob of the block named by pc_ToBlockName, or patch output when pc_ToBlockName is NULL.

pc_ToPartNum

Part number of the block input, block knob, or patch output specified by pc_ToBlockName and pc_ToPortName.

Notes

This signal carried by a connection created by this PatchCmd isn't guaranteed to reach the destination in the same frame that it was output if PATCH_CMD_SET_COHERENCE is set to FALSE.

Associated Files

<:audio:patch.h>

See Also

CreatePatchTemplate(), PatchCmd, ConnectPatchPorts(),
PATCH_CMD_ADD_TEMPLATE, PATCH_CMD_DEFINE_PORT, PATCH_CMD_DEFINE_KNOB,
PATCH_CMD_SET_CONSTANT, PATCH_CMD_SET_COHERENCE

PATCH_CMD_DEFINE_KNOB Defines a patch knob.**Synopsis**

```
typedef struct PatchCmdDefineKnob {
    uint32      pc_CmdID;           // PATCH_CMD_DEFINE_KNOB
    const char  *pc_KnobName;
    uint32      pc_NumParts;
    uint32      pc_KnobType;
    float32     pc_DefaultValue;
    uint32      pc_Pad5;
    uint32      pc_Pad6;
} PatchCmdDefineKnob;
```

Description

Define a knob or knob array for the patch. Once defined, ports and knobs can be connected to blocks added with PATCH_CMD_ADD_TEMPLATE or one another.

All patch knobs must be connected to something, or else the patch compiler returns an error.

Fields

pc_KnobName

Name for knob or knob array being added. Each port and knob in a patch must have a unique name. Names are compared case-insensitively. Port and knob names must not contain periods.

pc_NumParts

Number of parts for knob. 1 for a simple knob, >1 for a knob array.

pc_KnobType

Default knob type (AF_SIGNAL_TYPE_ value).

pc_DefaultValue

Default value for all parts of knob. This value is in whichever units are appropriate for the specified knob type.

Caveats

Doesn't currently detect unconnected patch knobs.

Associated Files

<:audio:patch.h>, <:audio:audio.h>

See Also

CreatePatchTemplate(), PatchCmd, DefinePatchKnob(),
PATCH_CMD_DEFINE_PORT, PATCH_CMD_EXPOSE, PATCH_CMD_CONNECT

PATCH_CMD_DEFINE_PORT

Defines a patch input or output port.

Synopsis

```
typedef struct PatchCmdDefinePort {  
    uint32      pc_CmdID;           // PATCH_CMD_DEFINE_PORT  
    const char *pc_PortName;  
    uint32      pc_NumParts;  
    uint32      pc_PortType;  
    uint32      pc_SignalType;  
    uint32      pc_Pad5;  
    uint32      pc_Pad6;  
} PatchCmdDefinePort;
```

Description

Define an input or output port or port array for the patch. Once defined, ports and knobs can be connected to blocks added with PATCH_CMD_ADD_TEMPLATE or one another.

All patch inputs and outputs must be connected to something, or else the patch compiler returns an error.

Fields

pc_PortName

Port name for port or port array being added. Each port and knob in a patch must have a unique name. Names are compared case-insensitively. Port and knob names must not contain periods.

pc_NumParts

Number of parts for port. 1 for a simple input or output, >1 for an input or output array.

pc_PortType

AF_PORT_TYPE_INPUT for an input or AF_PORT_TYPE_OUTPUT for an output.

pc_SignalType

Signal type (AF_SIGNAL_TYPE_ value).

Caveats

Doesn't currently detect unconnected patch inputs.

Associated Files

<:audio:patch.h>, <:audio:audio.h>

See Also

CreatePatchTemplate(), PatchCmd, DefinePatchPort(),
PATCH_CMD_DEFINE_KNOB, PATCH_CMD_EXPOSE, PATCH_CMD_CONNECT,

PATCH_CMD_END

Marks the end of a PatchCmd list.

Synopsis

```
typedef struct PatchCmdGeneric {  
    uint32      pc_CmdID;           // PATCH_CMD_END  
    uint32      pc_Args[6];  
} PatchCmdGeneric;
```

Description

This marks the end of a PatchCmd list.

Associated Files

<:audio:patch.h>

See Also

PatchCmd, NextPatchCmd(), PATCH_CMD_JUMP, PATCH_CMD_NOP

PATCH_CMD_EXPOSE

Exposes a patch port.

Synopsis

```
typedef struct PatchCmdExpose {  
    uint32      pc_CmdID;           // PATCH_CMD_EXPOSE  
    const char  *pc_PortName;  
    const char  *pc_SrcBlockName;  
    const char  *pc_SrcPortName;  
    uint32      pc_Pad4;  
    uint32      pc_Pad5;  
    uint32      pc_Pad6;  
} PatchCmdExpose;
```

Description

Expose a FIFO, envelope hook, or trigger belonging to one of the patch blocks, so that it may be accessed from outside the patch. For example, if `envelope.dsp` is one of the blocks in the patch, use this command to expose its envelope hook, `Env`, so that it may be attached to once the patch is compiled. Unless this is done, FIFOs, envelope hooks, and triggers remain private to the blocks of the patch and are not accessible to the clients of the finished patch.

Fields

`pc_PortName`

Name to give the exposed FIFO, envelope hook, or trigger when exposed. Ports, knobs, and exposed things all share the same name space and must have unique names. Names are compared case-insensitively. As with ports and knobs, exposed names cannot contain periods. May be the same name (or even the same pointer) as `pc_SrcPortName` as long as the above rules are satisfied.

`pc_SrcBlockName`

Name of the patch block containing the port to expose.

`pc_SrcPortName`

Name of the port within the specified patch block to expose.

Associated Files

<:audio:patch.h>

See Also

`CreatePatchTemplate()`, `PatchCmd`, `ExposePatchPort()`,
`PATCH_CMD_ADD_TEMPLATE`, `PATCH_CMD_DEFINE_PORT`, `PATCH_CMD_DEFINE_KNOB`

PATCH_CMD_JUMP

Links to another list of PatchCmds.

Synopsis

```
typedef struct PatchCmdJump {
    uint32      pc_CmdID;           // PATCH_CMD_JUMP
    const PatchCmd *pc_NextPatchCmd;
    uint32      pc_Pad2;
    uint32      pc_Pad3;
    uint32      pc_Pad4;
    uint32      pc_Pad5;
    uint32      pc_Pad6;
} PatchCmdJump;
```

Description

Links the end of one PatchCmd list to another.

Fields

`pc_NextPatchCmd`
Pointer to next PatchCmd list. Can be NULL, which causes this command to behave as a PATCH_CMD_END.

Associated Files

<:audio:patch.h>

See Also

PatchCmd, NextPatchCmd(), PATCH_CMD_NOP, PATCH_CMD_END

PATCH_CMD_NOP

Skip this command.

Synopsis

```
typedef struct PatchCmdGeneric {  
    uint32      pc_CmdID;           // PATCH_CMD_NOP  
    uint32      pc_Args[6];  
} PatchCmdGeneric;
```

Description

This command is skipped.

Associated Files

<:audio:patch.h>

See Also

PatchCmd, NextPatchCmd(), PATCH_CMD_JUMP, PATCH_CMD_END

PATCH_CMD_SET_COHERENCE Controls signal phase coherence along internal patch connections.

Synopsis

```
typedef struct PatchCmdSetCoherence {  
    uint32    pc_CmdID;           // PATCH_CMD_SET_COHERENCE  
    uint32    pc_State;           // TRUE to set, FALSE to clear  
    uint32    pc_Pad2;  
    uint32    pc_Pad3;  
    uint32    pc_Pad4;  
    uint32    pc_Pad5;  
    uint32    pc_Pad6;  
} PatchCmdSetCoherence;
```

Description

Controls whether the patch is to be built in such a way as to guarantee signal phase coherence along all internal connections (as defined by `PATCH_CMD_CONNECTs`). With `PATCH_CMD_SET_COHERENCE` set to `FALSE`, signals output from one constituent instrument may not propagate into the destination constituent instrument until the next audio frame.

The ability to control this aspect of patch construction is provided because guaranteeing signal phase coherence requires slightly more DSP code words and ticks than not doing so, and it is only necessary in situations where you need precise control over signal propagation delay (e.g., a comb filter). Patches default to being created with `PATCH_CMD_SET_COHERENCE` set to `TRUE`. The last `PATCH_CMD_SET_COHERENCE` encountered in a `PatchCmd` list determines the coherence of the entire patch.

Fields

`pc_State`

When set to `TRUE`, the default case, the patch is constructed with signal phase coherence guaranteed along internal connections. When set to `FALSE`, there is no guarantee of signal phase coherence.

Notes

To minimize propagation delay across a patch, instruments should be added to the patch (using `PATCH_CMD_ADD_TEMPLATE`) in the order of signal flow.

Associated Files

`<:audio:patch.h>`

See Also

`CreatePatchTemplate()`, `PatchCmd`, `SetPatchCoherence()`,
`PATCH_CMD_ADD_TEMPLATE`, `PATCH_CMD_CONNECT`

PATCH_CMD_SET_CONSTANT Sets a block input or knob to a constant.**Synopsis**

```
typedef struct PatchCmdSetConstant {  
    uint32      pc_CmdID;           // PATCH_CMD_SET_CONSTANT  
    const char  *pc_BlockName;  
    const char  *pc_PortName;  
    uint32      pc_PartNum;  
    float32     pc_ConstantValue;  
    uint32      pc_Pad5;  
    uint32      pc_Pad6;  
} PatchCmdSetConstant;
```

Description

Assigns a constant value to an internal unconnected input or knob part. This is a space-efficient way to set permanent values of things like coefficients, biases, frequencies, etc., that would otherwise require a knob.

Any block input or block knob can have up to one connection or constant. Unused block inputs default to 0; unused block knobs default to their default value.

Fields

pc_BlockName

Name of a patch block containing an input or knob to set to a constant. Patch outputs cannot be set to a constant, therefore NULL is illegal here.

pc_PortName

Name of an input or knob of the block named by pc_BlockName.

pc_PartNum

Part number of the input or knob specified by pc_BlockName and pc_PortName.

pc_ConstantValue

The constant value to set. The units depend on the thing being assigned. For knobs, the units are determined by the knob type (e.g. oscillator frequency in Hz, signed signal level, etc.). For inputs, this is always a signed signal in the range of -1.0 to 1.0.

Associated Files

<:audio:patch.h>

See Also

CreatePatchTemplate(), PatchCmd, SetPatchConstant(),
PATCH_CMD_ADD_TEMPLATE, PATCH_CMD_CONNECT

AddTemplateToPatch

Adds a PATCH_CMD_ADD_TEMPLATE PatchCmd to PatchCmdBuilder.

Synopsis

```
Err AddTemplateToPatch (PatchCmdBuilder *builder, const char
*blockName, Item insTemplate)
```

Description

Adds a PATCH_CMD_ADD_TEMPLATE PatchCmd to list under construction. This will add an Instrument Template Item with the specified name. Any Instrument Template may be added to the patch: standard DSP instruments, patches, or mixers. Multiple instances of the same Instrument Template Item may be added to any patch. Each template must be given a unique block name within the patch.

Arguments

builder

PatchCmdBuilder to add to.

blockName

Block name for template being added. Each block in a patch must have a unique name. Names are compared case-insensitively. Internal connections are specified using this block name. This string must remain valid for the life of the PatchCmdBuilder.

insTemplate

Instrument Template Item to use for this block. May be a standard DSP instrument template, a mixer, or a previously created patch. This Item must remain valid for the life of the PatchCmdBuilder.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Notes

To minimize propagation delay across a patch, instruments should be added in the order of signal flow.

Caveats

Any Items attached to insTemplate (i.e. Samples, Envelopes, or Tunings) are not propagated to the compiled patch.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_ADD_TEMPLATE, CreatePatchCmdBuilder(),
ExposePatchPort(), ConnectPatchPorts(), SetPatchConstant(),
SetPatchCoherence()

ConnectPatchPorts

Adds a PATCH_CMD_CONNECT PatchCmd to PatchCmdBuilder.

Synopsis

```
Err ConnectPatchPorts (PatchCmdBuilder *builder,  
                      const char *fromBlockName,  
                      const char *fromPortName,  
                      uint32 fromPartNum,  
                      const char *toBlockName,  
                      const char *toPortName,  
                      uint32 toPartNum)
```

Description

Adds a PATCH_CMD_CONNECT PatchCmd to list under construction. This will create internal signal connections between patch inputs, patch knobs, patch outputs, block inputs, block knobs, and block outputs.

Any block input or block knob can have up to one connection or constant. Unused block inputs default to 0; unused block knobs default to their default value. All patch outputs must be connected.

Arguments

builder

PatchCmdBuilder to add to.

fromBlockName

Name of a patch block containing an output to connect from. Or NULL to indicate that the source of the connection is a patch input or patch knob. (patch inputs and knobs appear as outputs for the purpose of internal connections) When non-NULL, this string must remain valid for the life of the PatchCmdBuilder.

fromPortName

Name of an output of the block named by fromBlockName, or patch input or patch knob when fromBlockName is NULL. This string must remain valid for the life of the PatchCmdBuilder.

fromPartNum

Part number of the block output, patch input, or patch knob specified by fromBlockName and fromPortName.

toBlockName

Name of a patch block containing an input or knob to connect to. Or NULL to indicate that the destination of the connection is a patch output. (patch outputs appear as inputs for the purpose of internal connections) When non-NULL, this string must remain valid for the life of the PatchCmdBuilder.

toPortName

Name of an input or knob of the block named by toBlockName, or patch output when toBlockName is NULL. This string must remain valid for the life of the PatchCmdBuilder.

toPartNum

Part number of the block input, block knob, or patch output specified by toBlockName and toPortName.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Notes

This signal carried by a connection created by this `PatchCmd` isn't guaranteed to reach the destination in the same frame that it was output if `PATCH_CMD_SET_COHERENCE` is set to `FALSE`.

Associated Files

<:audio:patch.h>, libc.a

See Also

`PatchCmd`, `PATCH_CMD_CONNECT`, `CreatePatchCmdBuilder()`,
`AddTemplateToPatch()`, `DefinePatchPort()`, `DefinePatchKnob()`,
`SetPatchConstant()`, `SetPatchCoherence()`

CreatePatchCmdBuilder

Creates a PatchCmdBuilder (convenience environment for constructing a PatchCmd List).

Synopsis

```
Err CreatePatchCmdBuilder (PatchCmdBuilder **resultBuilder)
```

Description

Creates an empty PatchCmdBuilder for constructing a PatchCmd list using the PatchCmd constructors (e.g. AddTemplateToPatch(), DefinePatchPort(), etc.). When the PatchCmd list is complete, pass the resulting PatchCmd list (returned by GetPatchCmdList()) to CreatePatchTemplate() to compile a Patch Template Item.

Because one typically calls a large number of PatchCmd constructors, it is inconvenient to check the success of each call. So when one of the constructors fails, the error code it returns is stored in the PatchCmdBuilder to prevent further constructors calls from succeeding: the original error code is returned by each subsequent constructor call.

GetPatchCmdBuilderError() also returns this error code. It may be called to check the validity of the entire PatchCmd list building process prior to using the PatchCmd list (see example below).

All pointers and Items added to the PatchCmdBuilder must remain valid for the life of the PatchCmdBuilder. The PatchCmdBuilder only needs to remain in existence until a patch is compiled from its PatchCmd list.

Arguments

resultBuilder

Pointer to a client-supplied buffer to receive a pointer to the allocated PatchCmdBuilder.

Return Value

Non-negative value on success, negative error code on failure.

When successful a pointer to the allocated PatchCmdBuilder is written to *resultBuilder. Nothing is written to this buffer on failure.

Implementation

Library call implemented in libc.a V29.

Example

```
Item MakePatchTemplate (void)
{
    Item tmpl_sawtooth = -1;
    PatchCmdBuilder *pb = NULL;
    Item result;

    // Load constituent templates
    if ((result = tmpl_sawtooth = LoadInsTemplate ("sawtooth.dsp",
    NULL)) < 0) goto clean;

    // Create PatchCmdBuilder
    if ((result = CreatePatchCmdBuilder (&pb)) < 0) goto clean;
```

```
        // Call PatchCmd constructors
        AddTemplateToPatch (pb, "saw", tmpl_sawtooth);
        DefinePatchPort (pb, "Output", 1, AF_PORT_TYPE_OUTPUT,
        AF_SIGNAL_TYPE_GENERIC_SIGNED);
        ConnectPatchPorts (pb, "saw", "Output", 0, NULL, "Output", 0);

        // See if an error occurred in one of the constructors
        if ((result = GetPatchCmdBuilderError (pb)) < 0) goto clean;

        // Create patch template
        result = CreatePatchTemplate (GetPatchCmdList(pb), NULL);

clean:
    DeletePatchCmdBuilder (pb);
    UnloadInsTemplate (tmpl_sawtooth);
    return result;
}
```

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, DeletePatchCmdBuilder(), GetPatchCmdList(),
GetPatchCmdBuilderError(), CreatePatchTemplate()

DefinePatchKnob

Adds a PATCH_CMD_DEFINE_KNOB PatchCmd to PatchCmdBuilder.

Synopsis

```
Err DefinePatchKnob (PatchCmdBuilder *builder, const char *knobName,  
                    uint32 numParts, uint32 knobType, float32  
                    defaultValue)
```

Description

Adds a PATCH_CMD_DEFINE_KNOB PatchCmd to list under construction. This will define a knob or knob array for the patch. Once defined, ports and knobs can be connected to blocks added with PATCH_CMD_ADD_TEMPLATE or one another.

All patch knobs must be connected to something, or else the patch compiler returns an error.

Arguments

builder

PatchCmdBuilder to add to.

knobName

Name for knob or knob array being added. Each port and knob in a patch must have a unique name. Names are compared case-insensitively. Port and knob names must not contain periods. This string must remain valid for the life of the PatchCmdBuilder.

numParts

Number of parts for knob. 1 for a simple knob, >1 for a knob array.

knobType

Default knob signal type (AF_SIGNAL_TYPE_ value).

defaultValue

Default value for all parts of knob. This value is in whichever units are appropriate for the specified knob type.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_DEFINE_KNOB, CreatePatchCmdBuilder(),
DefinePatchPort(), ExposePatchPort(), ConnectPatchPorts()

DefinePatchPort

Adds a PATCH_CMD_DEFINE_PORT PatchCmd to PatchCmdBuilder.

Synopsis

```
Err DefinePatchPort (PatchCmdBuilder *builder, const char *portName,  
                    uint32 numParts, uint32 portType, uint32  
                    signalType)
```

Description

Adds a PATCH_CMD_DEFINE_PORT PatchCmd to list under construction. This will define an input or output port or port array for the patch. Once defined, ports and knobs can be connected to blocks added with PATCH_CMD_ADD_TEMPLATE or one another.

All patch inputs and outputs must be connected to something, or else the patch compiler returns an error.

Arguments

builder
PatchCmdBuilder to add to.

portName
Port name for port or port array being added. Each port and knob in a patch must have a unique name. Names are compared case-insensitively. Port and knob names must not contain periods. This string must remain valid for the life of the PatchCmdBuilder.

numParts
Number of parts for port. 1 for a simple input or output, >1 for an input or output array.

portType
AF_PORT_TYPE_INPUT for an input or AF_PORT_TYPE_OUTPUT for an output.

signalType
Signal type (AF_SIGNAL_TYPE_ value).

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_DEFINE_PORT, CreatePatchCmdBuilder(),
DefinePatchKnob(), ExposePatchPort(), ConnectPatchPorts()

DeletePatchCmdBuilder

Deletes a PatchCmdBuilder created by
CreatePatchCmdBuilder().

Synopsis

```
void DeletePatchCmdBuilder (PatchCmdBuilder *builder)
```

Description

Deletes PatchCmdBuilder built by CreatePatchCmdBuilder() and all memory allocated by PatchCmd constructors. Does not delete any of the strings or Items placed in PatchCmds built in the PatchCmdBuilder.

Arguments

builder
Pointer to PatchCmdBuilder to delete. Can be NULL.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, CreatePatchCmdBuilder()

ExposePatchPort

Adds a PATCH_CMD_EXPOSE PatchCmd to PatchCmdBuilder.

Synopsis

```
Err ExposePatchPort (PatchCmdBuilder *builder, const char *portName,  
                    const char *srcBlockName, const char  
                    *srcPortName)
```

Description

Adds a PATCH_CMD_EXPOSE PatchCmd to list under construction. This will expose a FIFO, envelope hook, or trigger belonging to one of the patch blocks, so that it may be accessed from outside the patch. For example, if envelope.dsp is one of the blocks in the patch, use this command to expose its envelope hook, Env, so that it may be attached to once the patch is compiled. Unless this is done, FIFOs, envelope hooks, and triggers remain private to the blocks of the patch and are not accessible to the clients of the finished patch.

Arguments

builder

PatchCmdBuilder to add to.

portName

Name to give the exposed FIFO, envelope hook, or trigger when exposed. Ports, knobs, and exposed things all share the same name space and must have unique names. Names are compared case-insensitively. As with ports and knobs, exposed names cannot contain periods. May be the same name (or even the same pointer) as pc_SrcPortName as long as the above rules are satisfied. This string must remain valid for the life of the PatchCmdBuilder.

srcBlockName

Name of the patch block containing the port to expose. This string must remain valid for the life of the PatchCmdBuilder.

srcPortName

Name of the port within the specified patch block to expose. This string must remain valid for the life of the PatchCmdBuilder.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_EXPOSE, CreatePatchCmdBuilder(),
AddTemplateToPatch(), DefinePatchPort(), DefinePatchKnob()

GetPatchCmdBuilderError

Returns error code from a failed PatchCmd constructor.

Synopsis

```
Err GetPatchCmdBuilderError (const PatchCmdBuilder *builder)
```

Description

When one of the PatchCmd constructor functions (e.g., `AddTemplateToPatch()`, `DefinePatchPort()`, etc.) fails, all subsequent constructors also fail with the same error code. This function also returns that error code. If no PatchCmd constructors have failed, this function returns 0. When this function returns an error code, `GetPatchCmdList()` returns NULL.

You may use this function instead of checking the success of the constructor calls to simplify your code. See `CreatePatchCmdBuilder()` for an example.

Arguments

`builder`
Pointer to PatchCmdBuilder to test.

Return Value

The error code (a negative value) returned by the first failed PatchCmd constructor call in this PatchCmdBuilder, or 0 if no PatchCmd constructors have failed so far.

Implementation

Library call implemented in `libc.a V30`.

Associated Files

`<:audio:patch.h>`, `libc.a`

See Also

`CreatePatchCmdBuilder()`, `GetPatchCmdList()`

GetPatchCmdList

Returns PatchCmd list from a PatchCmdBuilder.

Synopsis

```
const PatchCmd *GetPatchCmdList (const PatchCmdBuilder *builder)
```

Description

Returns a pointer to the first PatchCmd in the list being constructed by the PatchCmdBuilder. Call this in order to get a PatchCmd list suitable for CreatePatchTemplate()

Arguments

builder
Pointer to PatchCmdBuilder to interrogate.

Return Value

Pointer to a PatchCmd list. Always returns a legal PatchCmd list for a valid PatchCmdBuilder, even if an empty one if called before any constructors are called.

Returns NULL if one of the PatchCmd constructors failed (when GetPatchCmdBuilderError() returns a negative value).

Apart from the NULL returned in case of a constructor failure, the address returned by this function is constant for the life of the PatchCmdBuilder.

Example

```
templateItem = CreatePatchTemplateVA (  
    GetPatchCmdList(builder),  
    TAG_END);
```

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, CreatePatchCmdBuilder(), CreatePatchTemplate(),
GetPatchCmdBuilderError()

SetPatchCoherence

Adds a PATCH_CMD_SET_COHERENCE PatchCmd to PatchCmdBuilder.

Synopsis

```
Err SetPatchCoherence (PatchCmdBuilder *builder, bool state)
```

Description

Adds a PATCH_CMD_SET_COHERENCE PatchCmd to list under construction. This controls whether the patch is to be built in such a way as to guarantee signal phase coherence along all internal connections. With coherence set to FALSE, signals output from one constituent instrument may not propagate into the destination constituent instrument until the next audio frame.

Arguments

builder

PatchCmdBuilder to add to.

state

When set to TRUE, the default case, the patch is constructed with signal phase coherence guaranteed along internal connections. When set to FALSE, there is no guarantee of signal phase coherence.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_SET_COHERENCE, CreatePatchCmdBuilder(),
AddTemplateToPatch(), ConnectPatchPorts()

SetPatchConstant

Adds a PATCH_CMD_SET_CONSTANT PatchCmd to PatchCmdBuilder.

Synopsis

```
Err SetPatchConstant (PatchCmdBuilder *builder, const char
*blockName,
                    const char *portName, uint32 partNum,
                    float32 constantValue)
```

Description

Adds a PATCH_CMD_SET_CONSTANT PatchCmd to list under construction. This will assign a constant value to an internal unconnected input or knob part. This is a space-efficient way to set permanent values of things like coefficients, biases, frequencies, etc., that would otherwise require a knob.

Any block input or block knob can have up to one connection or constant. Unused block inputs default to 0; unused block knobs default to their default value.

Arguments

builder

PatchCmdBuilder to add to.

blockName

Name of a patch block containing an input or knob to set to a constant. Patch outputs cannot be set to a constant, therefore NULL is illegal here. This string must remain valid for the life of the PatchCmdBuilder.

portName

Name of an input or knob of the block named by blockName. This string must remain valid for the life of the PatchCmdBuilder.

partNum

Part number of the input or knob specified by blockName and portName.

constantValue

The constant value to set. The units depend on the thing being assigned. For knobs, the units are determined by the knob type (e.g. oscillator frequency in Hz, signed signal level, etc.). For inputs, this is always a signed signal in the range of -1.0 to 1.0.

Return Value

Non-negative value on success, negative error code on failure.

Implementation

Library call implemented in libc.a V29.

Associated Files

<:audio:patch.h>, libc.a

See Also

PatchCmd, PATCH_CMD_SET_CONSTANT, CreatePatchCmdBuilder(),
AddTemplateToPatch(), ConnectPatchPorts()

Chapter 4

Beep Folio Calls

This section presents the reference documentation for the Beep folio and associated link libraries.

ConfigureBeepChannel

Configure a DMA channel.

Synopsis

```
Err ConfigureBeepChannel( uint32 channelNum, uint32 flags )
```

Description

Configure a Beep Machine DMA channel. This is used to control decompression and sample width.

Arguments

channelNum

Channel index. See Beep Machine document for information on the different channels available. The channelNum must be less than BEEP_NUM_CHANNELS. Note that only sample playing voices have an associated DMA channel.

flags

OR together these flags to configure DMA channel:

BEEP_F_CHAN_CONFIG_8BIT = 8 bit data

BEEP_F_CHAN_CONFIG_SQS2 = do SQS2 decompression

If you set BEEP_F_CHAN_CONFIG_SQS2 then you must also set BEEP_F_CHAN_CONFIG_8BIT.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

OpenBeepFolio()

See Also

StartBeepChannel()

GetBeepTime

Return the current Beep time.

Synopsis

```
uint32 GetBeepTime( void )
```

Description

Return the current Beep time. This is a count of the number of frames that have elapsed since the Beep Folio was first started. Each frame corresponds to the output of a stereo pair at 44100 Hz. Note that this is a 32 bit quantity that will wrap around approximately every 27 hours.

Arguments

none

Return Value

BeepTime in as an unsigned frame count.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

```
OpenBeepFolio()
```

See Also

```
GetAudioTime()
```


LoadBeepMachine

Load the Beep Machine DSP program into DSP

Synopsis

```
Item LoadBeepMachine( const char *machineName )
```

Description

Load the Beep Machine into the DSP. The machine name is defined in the *_machine.h include file as BEEP_MACHINE_NAME. The call to LoadBeepMachine() should, therefore, always be:

```
LoadBeepMachine( BEEP_MACHINE_NAME )
```

Only one Beep Machine can be loaded at a time. You must, therefore, call UnloadBeepMachine between calls to LoadBeepMachine().

Arguments

machineName

Set to BEEP_MACHINE_NAME name as defined in *_machine.h.

Return Value

The procedure returns an Item if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

```
OpenBeepFolio()
```

See Also

```
UnloadBeepMachine()
```

SetBeepChannelData

Specify data for a DMA channel.

Synopsis

```
Err SetBeepChannelData( uint32 channelNum, void *addr, int32
numSamples )
```

Description

Specify audio sample data block for a given DMA channel. When the channel is started, this data will be played. Once started, the data block must be reset with this function in order to replay the same data.

Arguments

channelNum

Channel index. See Beep Machine document for information on the different channels available. The channelNum must be less than BEEP_NUM_CHANNELS. Note that only sample playing voices have an associated DMA channel.

addr

Address of first data sample. 16 bit data must be 2 byte aligned. 8 bit data can be byte aligned.

numSamples

Number of samples in the block. Note that for 16 bit data this is the number of bytes divided by 2.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

OpenBeepFolio()

See Also

StartBeepChannel(), SetBeepChannelDataNext()

SetBeepChannelDataNext

Specify data to play after current data finishes.

Synopsis

```
Err SetBeepChannelDataNext( uint32 channelNum, void *addr, int32
numSamples,
    uint32 flags, int32 Signal )
```

Description

Specify audio sample data block to be played when the current block finishes. This is used to set up sample loops for continuous sound. It can also be used to chain sound buffers for spooling off of disc. Note that if you call this routine a second time, before the first data block has begun playing, then the first call will be ignored. To chain multiple sound buffers you must wait for each block to be started before calling `SetBeepChannelDataNext()` again. See `Examples/Audio/Beep/ta_spool.c` for an example.

Arguments

`channelNum`

Channel index. See Beep Machine document for information on the different channels available. The `channelNum` must be less than `BEEP_NUM_CHANNELS`. Note that only sample playing voices have an associated DMA channel.

`addr`

Address of first data sample. 16 bit data must be 2 byte aligned. 8 bit data can be byte aligned.

`numSamples`

Number of samples in the block. Note that for 16 bit data this is the number of bytes divided by 2.

`flags`

Flags for controlling sample playback.

`BEEP_F_IF_GO_FOREVER` = play this block over and over. If not set then it will play this block once and stop.

`signal`

Request that the calling task be sent this signal when this block has begun playing. At that time you may specify a new block to be played after this one finishes. This can be used to spool multiple blocks of data.

You can disable the request by passing zero for the value of `Signal`. One can pass `Addr = NULL` to cancel a `DataNext` request. The DMA will then stop after the current block.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, `System.m2/Modules/beep`

Module Open Requirements

`OpenBeepFolio()`

See Also

`StartBeepChannel()`, `SetBeepChannelData()` [Examples/Audio/Beep/ta_spool.c](#)

SetBeepParameter

Set a global Beep parameter.

Synopsis

```
Err SetBeepParameter ( uint32 ParameterID, float32 Value )
```

Description

Set a global Beep parameter.

Arguments

ParameterID

Identifies parameter. Defined in the beep machines include file.

Value

Value to set parameter to. See Beep Machine document for information on the range of this Value.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

OpenBeepFolio()

See Also

SetBeepVoiceParameter()

SetBeepVoiceParameter

Set a Beep voice parameter.

Synopsis

```
Err SetBeepVoiceParameter ( uint32 voiceNum,  
                             uint32 parameterID, float32 value )
```

Description

Set a Beep parameter specific to a single voice.

Arguments

voiceNum

Voice index. See Beep Machine document for information on the different voices available.

parameterID

Identifies parameter. Defined in the beep machines include file.

value

Value to set parameter to. See Beep Machine document for information on the range of this Value.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

OpenBeepFolio()

See Also

SetBeepParameter()

StartBeepChannel

Start a DMA channel.

Synopsis

```
Err StartBeepChannel( uint32 channelNum )
```

Description

Enable DMA so that the current sample block begins playing. If DMA has been disabled by calling `StopBeepChannel()`, then the playback will resume where it left off. To restart the sample at the beginning, you must call `SetBeepChannelData()`.

Arguments

`channelNum`

Channel index. See Beep Machine document for information on the different channels available.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

`OpenBeepFolio()`

See Also

`StopBeepChannel()`, `SetBeepChannelData()`

StopBeepChannel

Stop a DMA channel.

Synopsis

```
Err StopBeepChannel( uint32 channelNum )
```

Description

Disable DMA on the specified channel.

Arguments

channelNum

Channel index. See Beep Machine document for information on the different channels available.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

```
OpenBeepFolio()
```

See Also

```
StartBeepChannel(), SetBeepChannelData()
```


UnloadBeepMachine

Unload the Beep Machine from the DSP

Synopsis

```
Err UnloadBeepMachine( beepMachine )
```

Description

Unload the Beep Machine into the DSP.

Arguments

beepMachine

Item returned by LoadBeepMachine.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

V30.

Associated Files

<:beep:beep.h>, System.m2/Modules/beep

Module Open Requirements

OpenBeepFolio()

See Also

LoadBeepMachine()

basic.bm

Basic Beep Machine.

Function

Basic 32 channel sample player plus filtered noise.

This beep machine provides 33 voices of two type. There are 32 voices of the first type which provides sample playback with envelope control over the amplitude. Each of these voices has an associated DMA channel. Hardware SQS2 decompression can be turned on for any channel. Each voice also has an LFO which can be used to modulate SampleRate and Amplitude.

There is 1 voice of the second type which provides a filtered noise source.

Common Voice Parameters**BMVP_AMPLITUDE - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED**

Amplitude of voice. Note that an envelope is used to smooth amplitude changes to avoid pops. The amplitude of the voice will thus move smoothly to this value at the rate specified using BMVP_AMPLITUDE_RATE. Range is -1.0 to 1.0. Defaults to 0.0.

BMVP_AMPLITUDE_RATE - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Controls envelope rate at which the amplitude will change from its current value to the value last specified using BMVP_AMPLITUDE. A value of 1.0 will allow the Amplitude to change almost instantaneously. Range is 0.0 to 1.0. Defaults to 0.002.

$$\text{Rate} = 1.0 / (44100.0 * \text{Time_In_Seconds})$$

BMVP_LEFT_GAIN - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED BMVP_RIGHT_GAIN - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Control the amount of this voice to appear in the left or right audio output channel. This controls an internal mixer. There is no envelope to smooth out changes of these parameters so a sudden change in value could cause an audible pop. An application must set these gains carefully. If you set these too low, the audio in the title will be very quiet. If you set them too high then the combined out put of the voices could exceed the maximum limit and be clipped. Clipping would cause harsh noise whenever the sound is loud. The total signal level in either the left or right side cannot exceed +/- 1.0. Range is -1.0 to 1.0. Defaults to 0.125.

BMVP_LFO_RATE - AUDIO_SIGNAL_TYPE_OSC_FREQ

Controls the frequency of the voices LFO. Range is -2756.0 to +2756.0 Hz. Default is 1.0.

BMVP_AMP_MOD_DEPTH - AUDIO_SIGNAL_GENERIC_SIGNED

Controls the amount that the LFO modulates the Amplitude. mixer. Range is -1.0 to 1.0. Defaults to 0.0.

Parameters specific to Sample Player Voices #0 to 31**BMVP_SAMPLE_RATE - AUDIO_SIGNAL_TYPE_SAMPLE_RATE**

Controls the playback rate for audio data on the associated channel. Range is 0.0 to 88200.0. Default is 44100.0.

BMVP_FREQ_MOD_DEPTH - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

Controls the amount that the LFO modulates the SampleRate. Range is Range is -44100.0 to 44100.0. Default is 0.0.

Parameters specific to Filtered Noise Voice #32

BMVP_FREQ_MOD_DEPTH - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

Controls the amount that the LFO modulates the Filter Cutoff. Range is -1.0 to 1.0.
Default is 0.0.

BMVP_FILTER_CUTOFF - AUDIO_SIGNAL_GENERIC_SIGNED

Controls the filter cutoff frequency. Range is -1.0 to 1.0. Defaults to 0.25. See the `svfilter.dsp` Frequency knob for more information.

BMVP_CUTOFF_MOD_DEPTH - AUDIO_SIGNAL_GENERIC_SIGNED

Controls the amount that the LFO modulates the Filter Cutoff. Range is -1.0 to 1.0.
Default is 0.0.

BM_NUM_CHANNELS (32)

BM_NUM_VOICES (33)

Implementation

V30

See Also

`LoadBeepMachine()`

Chapter 5

DSP Instruments

This section presents the reference documentation for the Audio DSP instruments.

--DSP-Instrument-Overview--

Overview of DSP Instrument documentation.

Instrument Categories

All DSP instruments are grouped into one of the following categories:

Accumulator

Instruments which perform a mathematical operation on their inputs and the DSP accumulator, and then leave their results in the DSP accumulator (e.g., `add_accum.dsp`). These instrument templates may only be used within a patch template. They may not be used as stand-alone instruments.

Arithmetic

Instruments which perform a mathematical operation on their inputs (e.g., `add.dsp`). All of these instruments produce static results for static inputs.

Control_Signal

Instruments which output a time-variant signal which may be used to control other signals. This set includes low-frequency oscillators (e.g., `triangle_lfo.dsp`), signal analysis instruments (e.g., `envfollower.dsp`), and other miscellaneous time-variant operations.

Diagnostic

Instruments which may be useful during development of an title, but are not intended for use in a released title.

Effects

Instruments which operate on an audio signal to produce some special sound effect (e.g., `cubic_amplifier.dsp`, `svfilter.dsp`, `delay_f1.dsp`).

Line_In_And_Out

Instruments which provide access to the audio Line In and Line Out connections (e.g., `line_out.dsp`).

Sampled_Sound

Instruments which play sampled sounds (represented by `Sample` items) by reading from a DMA channel (e.g., `sampler_16_v1.dsp`).

Sound_Synthesis

Instruments which artificially create audio waveforms. There is a fairly conventional set of analog-style oscillators (e.g., `sawtooth.dsp`, `pulse.dsp`) and a few less conventional sound sources.

Ports

Instrument ports are documented by port type in the following sections:

AF_PORT_TYPE_INPUT - Inputs

Unless otherwise noted, all of these are `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`, single-part, and have a default value of 0.0.

AF_PORT_TYPE_OUTPUT - Outputs

Unless otherwise noted, all of these are `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED` and single-part.

AF_PORT_TYPE_KNOB - Knobs

Unless otherwise noted all of these are single-part. The signal types and default value of these vary depending on function, therefore this information is always indicated.

AF_PORT_TYPE_IN_FIFO - Input FIFOs

AF_PORT_TYPE_OUT_FIFO - Output FIFOs

AF_PORT_TYPE_TRIGGER - Triggers

AF_PORT_TYPE_ENVELOPE - Envelope Hooks

Resources

DSP resource usage is noted in the Resources section of each instrument's documentation. Unless otherwise noted, the amount quoted is per instrument. Some instruments take advantage of shared code and data, and therefore only the first allocation of such an instrument requires allocation of the shared code and data. This is noted where applicable.

Tick usage amounts are per frame. Full-rate instruments will consume eight times the quoted number of ticks from each eight-frame batch. Half-rate instruments consume four times the quoted number of ticks from each eight-frame batch. Eighth-rate instruments consume just the quoted number of ticks from each eight-frame batch.

Note: Resource information is listed to help you choose instruments based on their impact on the available DSP resources. We reserve the right to reduce the amount of resources required by any instrument in future releases of the Portfolio operating system.

See Also

`LoadInsTemplate(), CreateMixerTemplate(), CreatePatchTemplate(),
Template, PatchCmd, --Audio-Port-Types--, --Audio-Signal-Types--`

Mixer

General description of custom mixer Templates built by `CreateMixerTemplate()`.

Description

Mixes DSP audio signals in a manner similar to an audio mixing board. Because of the enormous range of possible configurations, there isn't any way to satisfy everyone with a set of statically defined mixer instruments. So mixers are constructed on demand based on a simple set of parameters:

- Number of Inputs
- Number of Outputs
- Whether there should be a master Amplitude knob
- Whether to connect directly to line out or to an Output port

This information is packed into a 32-bit `MixerSpec` (defined in `<:audio:audio.h>`) using the macro `MakeMixerSpec()`.

Mixers can also be viewed as a matrix multiplication of the form:

$$\text{Output}[m] = \text{Amplitude} * (\text{Input}[0]*\text{Gain}[0,m] + \text{Input}[1]*\text{Gain}[1,m] + \dots)$$

where:

n is in the range of 0..numInputs-1

m is in the range of 0..numOutputs-1

The following is the generic documentation for all mixers where numInputs is the number of inputs, and numOutputs is the number of outputs.

Knobs

Gain - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`, numInputs*numOutput parts

The Gain knob is a two-dimensional matrix of gain coefficients indexed by input channel number and output channel number. Use the macro `CalcMixerGainPart()` to translate a two-dimensional gain knob reference to a one-dimensional knob part suitable for `SetKnobPart()`.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

This is global amplitude knob which affects all output channels after they have been mixed via the Gain knob parts. This knob is present only when `AF_F_MIXER_WITH_AMPLITUDE` is set.

Inputs

Input - numInput parts

Outputs

Output - numOutput parts

This port is present only when `AF_F_MIXER_WITH_LINE_OUT` is not specified. When `AF_F_MIXER_WITH_LINE_OUT` is specified the mixer's output is automatically routed to the DAC just as if the Output had been connected to `line_out.dsp`.

Resources

Ticks: variable

Code: variable

Data: variable

Implementation

V28

Associated Files

`<:audio:audio.h>`

See Also

`MakeMixerSpec()`, `CreateMixerTemplate()`, `line_out.dsp`

add_accum.dsp

Adds signed signal to DSP accumulator.

Description

This instrument performs a signed addition between its input and the DSP accumulator: $\text{Accum} = \text{Accum} + \text{Input}$. The result is clipped into range, and left in the accumulator.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed input value. -1.0 to 1.0. Defaults to 0.0.

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

See Also

multiply_accum.dsp, subtract_accum.dsp, input_accum.dsp,
output_accum.dsp, add.dsp

input_accum.dsp

Loads the accumulator from an input.

Description

This instrument places the input value into the accumulator ($\text{Accum} = \text{Input}$), where it can be subsequently used by other instruments in the Accumulator group.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed input value. -1.0 to 1.0. Defaults to 0.0

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

See Also

`output_accum.dsp`, `add_accum.dsp`, `multiply_accum.dsp`, `subtract_accum.dsp`

multiply_accum.dsp

Multiplies accumulator by a signed signal.

Description

This is a 4-quadrant multiplication: $\text{Accum} = \text{Accum} * \text{Input}$.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Signed factor in the range of -1.0 to 1.0. Defaults to 1.0.

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

Caveats

Multiplying -1.0 by -1.0 causes an overflow yielding a result of -1.0. Thus, if you multiply a signal by -1.0, it will pop if the signal also contains -1.0.

See Also

add_accum.dsp, subtract_accum.dsp, input_accum.dsp, output_accum.dsp,
multiply.dsp

output_accum.dsp

Stores accumulator to an output.

Description

This instrument passes the accumulator value to the output: $\text{Output} = \text{Accum}$.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Outputs

Output
Signed output value. -1.0 to 1.0.

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

See Also

`input_accum.dsp`, `add_accum.dsp`, `multiply_accum.dsp`, `subtract_accum.dsp`

subtract_accum.dsp

Subtracts accumulator from input.

Description

Subtracts the accumulator value from the input, leaving the result in the accumulator: $\text{Accum} = \text{Input} - \text{Accum}$.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed input value. -1.0 to 1.0. Defaults to 0.0.

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

See Also

add_accum.dsp, multiply_accum.dsp, subtract_from_accum.dsp,
input_accum.dsp, output_accum.dsp, subtract.dsp

subtract_from_accum.dsp

Subtracts input from accumulator.

Description

Subtracts the input value from the accumulator, leaving the result in the accumulator: $\text{Accum} = \text{Accum} - \text{Input}$.

This instrument can only be used within a patch. It may not be used as a stand-alone instrument.

Knobs

Input - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`
Signed input value. -1.0 to 1.0. Defaults to 0.0.

Resources

Ticks: 4

Code: 4 words

Data: 1 word

Implementation

V30

See Also

`add_accum.dsp`, `multiply_accum.dsp`, `subtract_accum.dsp`, `input_accum.dsp`, `output_accum.dsp`, `subtract.dsp`

add.dsp

Adds two signed signals.

Description

This instrument performs a signed addition between its two inputs. The result is clipped into range.

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed input values. -1.0 to 1.0. Defaults to 0.0

Outputs

Output
Signed, clipped sum of InputA and InputB. -1.0 to 1.0.

Resources

Ticks: 6

Code: 6 words

Data: 3 words

Implementation

V24

See Also

multiply.dsp, subtract.dsp, timesplus.dsp, add_accum.dsp

expmod_unsigned.dsp

Exponential modulation of an unsigned signal.

Description

Implements the function $\text{Input} \times (2.0^{**}(\text{Modulation}))$.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED

Unsigned scalar in the range of 0.0 to 2.0. Defaults to 1.0.

Modulation - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Signed exponent in the range of -1.0 to 1.0. Defaults to 0.0.

Outputs

Output - AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED

$\text{Input} \times (2.0^{**}(\text{Modulation}))$ clipped to the range of 0.0 to 2.0.

Resources

Ticks: 31

Code: 29 words

Data: 4 words

Implementation

V28

Caveats

Multiplication can result in a value greater than 2.0 which will be clipped to 2.0.

See Also

`add.dsp`, `multiply.dsp`

latch.dsp

Passes its input to output if gate held above zero.

Description

This instrument acts as a wire if the Gate input is greater than zero. If the gate is less than or equal to zero, then the output is held steady. This can be used basically a sample and hold circuit.

Knobs

Gate - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
If greater than zero, signal passes, otherwise held.

Inputs

Input
-1.0 to +1.0

Outputs

Output
-1.0 to +1.0

Resources

Ticks: 9

Code: 7 words

Data: 2 words

Implementation

V27

See Also

`noise.dsp`, `impulse.dsp`, `randomhold.dsp`

maximum.dsp

Picks the maximum of two input signals.

Description

Outputs the larger of InputA and InputB. This can be used for clipping.

By connecting Output to InputB, you can measure a historical maximum. In this configuration, `StartInstrument()` can be used to reset the historical maximum.

Knobs

InputA, InputB - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`
Signed value -1.0 to 1.0. Defaults to 0.0

Outputs

Output
Signed result of `MAX(InputA, InputB)` in the range of -1.0 to 1.0.

Resources

Ticks: 13

Code: 11 words

Data: 3 words

Implementation

V24

See Also

`minimum.dsp`, `add.dsp`

minimum.dsp

Picks the minimum of two input signals.

Description

Outputs the smaller of InputA and InputB. This can be used for clipping.

By connecting Output to InputB, you can measure a historical minimum. In this configuration, `StartInstrument()` can be used to reset the historical maximum.

Knobs

InputA, InputB - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`
Signed value -1.0 to 1.0. Defaults to 0.0

Outputs

Output
Signed result of `MIN(InputA, InputB)` in the range of -1.0 to 1.0.

Resources

Ticks: 13

Code: 11 words

Data: 3 words

Implementation

V24

See Also

`maximum.dsp`, `add.dsp`

multiply.dsp

Multiplies two signed input signals (ring modulator).

Description

This is a 4-quadrant multiplier. It can be used as a single-channel mixer (gain * signal) or as a ring modulator (signal1 * signal2).

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed inputs in the range of -1.0 to 1.0. Defaults to 1.0.

Outputs

Output
Signed result of InputA * InputB in the range of -1.0 to 1.0.

Resources

Ticks: 6

Code: 6 words

Data: 3 words

Implementation

V20

Caveats

Multiplying -1.0 by -1.0 causes an overflow yielding a result of -1.0. Thus, if you multiply a signal by -1.0, it will pop if the signal also contains -1.0.

See Also

`multiply_unsigned.dsp`, `multiply_accum.dsp`, `timesplus.dsp`, Mixer

multiply_unsigned.dsp

Multiplies two unsigned signals.

Description

This is a 1-quadrant multiplier.

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED
Unsigned inputs in the range of 0.0 to 2.0. Defaults to 1.0.

Outputs

Output - AUDIO_SIGNAL_TYPE_GENERIC_UNSIGNED
Unsigned result of InputA * InputB in the range of 0.0 to 2.0.

Resources

Ticks: 13

Code: 12 words

Data: 3 words

Implementation

V28

Caveats

Multiplication can result in a value greater than 2.0 which will be clipped to 2.0.

See Also

multiply.dsp, timesplus.dsp

schmidt_trigger.dsp

Comparator with hysteresis and trigger.

Description

Comparator with hysteresis. Outputs boolean value: TRUE when Input rises above SetLevel, FALSE when Input equals or falls below ResetLevel. When input value goes below ResetLevel and then back above SetLevel, it will trigger a CPU interrupt if `ArmTrigger()` has been called.

Knobs

SetLevel - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Set Output to TRUE when Input rises ABOVE SetLevel. -1.0 to 1.0, defaults to 0.

ResetLevel - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Set Output to FALSE when Input EQUALS or goes BELOW ResetLevel. -1.0 to 1.0, defaults to 0.

Inputs

Input

Signal to compare against set and reset levels.

Outputs

Output (Boolean)

TRUE (>0) or FALSE (0). Output is reset to FALSE whenever the instrument is started.

Triggers

Trigger

Sent when instrument's Output goes from FALSE to TRUE.

Resources

Ticks: 20

Code: 20 words

Data: 3 words

Triggers: 1

Implementation

V27

Caveats

Starting this instrument with Input > SetLevel causes the Output to go TRUE as soon as the instrument runs. Because the Output is first reset to FALSE when started, this FALSE->TRUE transition causes the trigger to be sent (just as any other FALSE->TRUE transition of the Output) as soon as the instrument runs. This happens regardless of whether Input ever went <= ResetLevel.

See Also

`ArmTrigger()`, `DisarmTrigger()`

subtract.dsp

Returns the difference between two signed signals.

Description

Outputs $\text{InputA} - \text{InputB}$. The result is clipped into the range of -1.0 and 1.0.

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed input values. -1.0 to 1.0. Defaults to 0.0.

Outputs

Output

Signed, clipped result of $\text{InputA} - \text{InputB}$. -1.0 to 1.0.

Resources

Ticks: 6

Code: 6 words

Data: 3 words

Implementation

V24

See Also

`add.dsp`, `timesplus.dsp`, `subtract_accum.dsp`, `subtract_from_accum.dsp`

timesplus.dspSingle-operation multiply and accumulate ($A*B+C$).**Description**

This instrument returns the result of $\text{InputA} * \text{InputB} + \text{InputC}$. This is useful for mixing control signals. The signed result is clipped to -1.0 to 1.0.

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed factors. -1.0 to 1.0, defaults to 1.0.

InputC - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Signed offset. -1.0 to 1.0, defaults to 0.0.

Outputs

Output
Clipped, signed result of $\text{InputA} * \text{InputB} + \text{InputC}$. -1.0 to 1.0.

Resources

Ticks: 7

Code: 7 words

Data: 4 words

Implementation

V20

Caveats

Multiplying -1.0 by -1.0 causes an overflow yielding a result of -1.0. Thus, if you multiply a signal by -1.0, it will pop if the signal also contains -1.0.

Because the result is signed, this instrument's use is limited for operating on sample rates, which is unsigned. `multiply_unsigned.dsp` and `expmod_unsigned.dsp` are good unsigned alternatives.

See Also

`add.dsp`, `multiply.dsp`, `timesplus_noclip.dsp`, `multiply_unsigned.dsp`,
`expmod_unsigned.dsp`

timesplus_noclip.dsp

Single-operation, unclipped multiply and accumulate ($A*B+C$).

Description

This instrument returns the result of $\text{InputA} * \text{InputB} + \text{InputC}$. This is useful for mixing control signals. The signed result is not clipped.

Knobs

InputA, InputB - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Signed factors. -1.0 to 1.0, defaults to 1.0.

InputC - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED -

Signed offset. -1.0 to 1.0, defaults to 0.0.

Outputs

Output

Signed result of $\text{InputA} * \text{InputB} + \text{InputC}$. -1.0 to 1.0.

Resources

Ticks: 7

Code: 7 words

Data: 4 words

Implementation

V29

Caveats

Multiplying -1.0 by -1.0 causes an overflow yielding a result of -1.0. Thus, if you multiply a signal by -1.0, it will pop if the signal also contains -1.0.

See Also

add.dsp, multiply.dsp, timesplus.dsp, multiply_unsigned.dsp,
expmod_unsigned.dsp

times_256.dsp

Fast multiplication by 256.

Description

This instrument provides a fast multiplication by 256.

Inputs

Input
-1.0 to +1.0.

Outputs

Output
Unclipped result of Input * 256 in the range of -1.0 to +1.0.

Resources

Ticks: 5

Code: 5 words

Data: 1 word

Implementation

V30

See Also

`multiply.dsp`

envelope.dsp

Interpolate a segment of an Envelope.

Description

This instrument is used by the audio folio to produce multi-segmented Envelopes. The main CPU passes each segment to this DSP instrument, which interpolates the values between the envelope points. This can be used to produce contours to smoothly control amplitude or other parameters. See the audio folio function `CreateEnvelope()` and the example program `simple_envelope`. The knobs are controlled internally by the audio folio to produce the desired effect.

You can also use this instrument by itself to generate ramp functions. When you change the `Env.request` knob, the envelope will start changing from its current output value toward the value of `Env.request`. An internal phase value will go from 0.0 to 1.0 at a rate controlled by `Env.incr`. When the internal phase reaches 1.0, the output will equal `Env.request`.

Knobs

`Env.request` - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Target value. -1.0 to 1.0. Defaults to 0.0. Changing the value of this knob resets the internal phase to zero.

`Env.incr` - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Amount to increment internal phase per execution frame. A higher number means it will reach `Env.request` more quickly. Range is 0.0 to 1.0, defaults to 0.0005.

`Amplitude` - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0. Defaults to 1.0

Envelope Hooks

`Env`

Envelope hook. This envelope accepts signed envelope data.

Outputs

`Output`

Resources

Ticks: 29

Code: 4 words per instrument + 22 words shared overhead

Data: 8 words

Implementation

V20

Notes

To use with an Envelope Item, attach the Envelope to the hook "Env" by calling `call CreateAttachment()`. To control this directly, use the knobs "Env.request" and "Env.incr". While an Envelope is attached to this instrument, the knobs should not be controlled directly.

See Also

`Envelope`, `CreateEnvelope()`, `CreateAttachment()`, `simple_envelope`

envfollower.dsp

Tracks the contour of a signal.

Description

This instrument tracks the positive peaks of an input signal. It decays exponentially based on the droop factor. It outputs a fairly smooth signal that can be used to control other signals.

Knobs

Droop - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Number to multiply previous result by to generate exponential decay. Higher numbers give slower decay. 0.0 to 1.0. Default is 0.9995.

Inputs

Input

Signed signal to analyze.

Outputs

Output

Contour of signal.

Resources

Ticks: 16

Code: 15 words

Data: 3 words

Implementation

V20

See Also

`envelope.dsp`, `minimum.dsp`, `maximum.dsp`, `schmidt_trigger.dsp`

integrator.dsp

Integrates an input signal (ramp generator).

Description

Adds its input to an internal accumulator. Accumulator is clipped at -1.0 and 1.0. Output can be scaled by an Amplitude and Offset so this instrument can be used as a simple ramp, or envelope generator. Give it a positive Input to make it go up and a negative Input to make it go down.

This would typically be used by connecting the Output of the integrator to the Amplitude knob of a sampler that you wanted to turn on or off. To slowly turn up the sampler, set the Input of the integrator to a small positive value, like 0.001. To slowly turn down the sampler, set the Input of the integrator to a small negative value. This could be particularly useful in streaming applications. This instrument requires less DSP resources than `envelope.dsp`.

If Amplitude=0.5 and Offset=0.5, then Output will range from 0.0 to 1.0.

If Amplitude=1.0 and Offset=1.0, then Output will range from 0.0 to 2.0.

If Amplitude=0.25 and Offset=0.25, then Output will range from 0.0 to 0.5.

The following iterative formula is used:

Accumulator = CLIP(Accumulator + Input)

Output = (Accumulator * Amplitude) + Offset

Whenever this instrument is started, the accumulator is set to -1.0. This will result in an output of 0.0 if the Amplitude and Offset are set to their default values.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to +1.0, defaults to 0.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to +1.0, defaults to 0.5.

Offset - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to +1.0, defaults to 0.5.

Outputs

Output
-2.0 to +2.0 depending on Amplitude and Offset (result of multiply- accumulate isn't clipped).

Resources

Ticks: 9

Code: 9 words

Data: 5 words

Implementation

V28

See Also

`envelope.dsp`, `slew_rate_limiter.dsp`, `sawtooth.dsp`

pulse_lfo.dsp

Pulse wave Low-Frequency Oscillator.

Description

This instrument is a pulse wave generator that uses extended precision arithmetic to give lower frequencies than `pulse.dsp`. It also has better resolution at the same frequency. The frequency of this instrument is 256 times lower than its corresponding high frequency version. It is useful as a modulation source for controlling other instruments, or for bass instruments.

Knobs

Frequency - `AUDIO_SIGNAL_TYPE_LFO_FREQ`

Oscillator frequency in Hertz. Ranges from -86.1 Hz to +86.1 Hz. Default is 17.2 Hz.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

-1.0 to +1.0, defaults to 1.0.

PulseWidth - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

Level for comparator used to generate pulse wave. 0 gives a 50% duty cycle pulse wave.

Positive values cause the positive portion of the pulse to be narrower than the negative.

Negative values cause the negative portion of the pulse to be narrower than the positive.

-1.0 to +1.0, defaults to 0.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 19

Code: 18 words

Data: 6 words

Implementation

V21

See Also

`square_lfo.dsp`, `triangle_lfo.dsp`, `pulse.dsp`

randomhold.dsp

Generates random values and holds them.

Description

This instrument generates new random numbers at a given rate and holds steady until a new number is chosen. Think of it as a random staircase function. This is handy for weird synthetic sound effects (e.g., random frequency values or pitch modulation of an oscillator).

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Sample-and-hold frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Controls how frequently a new random value is chosen.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 16

Code: 13 words

Data: 5 words

Implementation

V21

See Also

`latch.dsp`, `noise.dsp`, `rednoise.dsp`

rednoise_lfo.dsp

Red noise LFO generator.

Description

This instrument interpolates straight line segments between pseudo-random numbers to produce "red" noise. It is useful as a slowly changing random control generator for natural sounds. This is a low-frequency version of `rednoise.dsp`.

Knobs

Frequency - `AUDIO_SIGNAL_TYPE_LFO_FREQ`

Frequency in Hertz. Ranges from -86.1 Hz to +86.1 Hz. Default is 17.2 Hz. Controls how frequently a new random value is chosen.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 33

Code: 34 words

Data: 7 words

Implementation

V27

See Also

`noise.dsp`, `randomhold.dsp`, `rednoise.dsp`, `triangle_lfo.dsp`

slew_rate_limiter.dsp

Slew rate limiter.

Description

The Output of this instrument tracks its Input but can only change by a maximum amount. This is useful for slowing down control signals like Amplitude that can produce pops is changed too quickly. It can also be used to produce glide effects.

Knobs

Input - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to +1.0, defaults to 0.0.

Rate - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
The output can change by no more than +/- Rate. in a single frame. -1.0 to +1.0, defaults to 1.0.

Outputs

Output
-1.0 to +1.0

Resources

Ticks: 19

Code: 19 words

Data: 3 words

Implementation

V28

See Also

`envelope.dsp`, `integrator.dsp`

square_lfo.dsp

Square wave Low-Frequency Oscillator.

Description

This instrument is a square wave generator that uses extended precision arithmetic to give lower frequencies than `square.dsp`. It also has better resolution at the same frequency. The frequency of this instrument is 256 times lower than its corresponding high frequency version. It is useful as a modulation source for controlling other instruments, or for bass instruments.

Knobs

Frequency - `AUDIO_SIGNAL_TYPE_LFO_FREQ`

Oscillator frequency in Hertz. Ranges from -86.1 Hz to +86.1 Hz. Default is 17.2 Hz.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 17

Code: 16 words

Data: 5 words

Implementation

V21

See Also

`pulse_lfo.dsp`, `triangle_lfo.dsp`, `square.dsp`

triangle_lfo.dsp

Triangle wave Low-Frequency Oscillator.

Description

This instrument is a triangle wave generator that uses extended precision arithmetic to give lower frequencies than `triangle.dsp`. It also has better resolution at the same frequency. The frequency of this instrument is 256 times lower than its corresponding high frequency version. It is useful as a modulation source for controlling other instruments, or for bass instruments.

Knobs

Frequency - `AUDIO_SIGNAL_TYPE_LFO_FREQ`

Oscillator frequency in Hertz. Ranges from -86.1 Hz to +86.1 Hz. Default is 17.2 Hz.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 24

Code: 22 words

Data: 5 words

Implementation

V21

See Also

`pulse_lfo.dsp`, `square_lfo.dsp`, `triangle.dsp`

benchmark.dsp

Outputs current DSPP tick count.

Description

Output current DSPP tick count. Used by dspfaders for accurate benchmarking of DSP instruments. Do not use in a title.

Outputs

Output

Number of elapsed ticks since head instrument sampled tick clock divided by 32768.0.
Value in the range of 0.0 to 32767.0 / 32768.0.

Resources

Ticks: 6

Code: 6 words

Data: 1 word

Implementation

V24

Caveats

This is a diagnostic instrument. Do not use in a title.

The Output has signal type `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED` instead of `AUDIO_SIGNAL_TYPE_WHOLE_NUMBER`, which would return ticks as integer values. If you Probe this output, create the Probe with `{ AF_TAG_TYPE, AUDIO_SIGNAL_TYPE_WHOLE_NUMBER }`.

cubic_amplifier.dsp

Non-linear amplifier (distortion effect).

Description

This instrument amplifies an incoming signal using a cubic function that results in distortion similar to a guitar fuzz box. For low level signals, the gain is approximately 3X.

The gain function is:

if (Input < 0.0) Output = (Input + 1.0)**3 - 1.0;

else Output = (Input - 1.0)**3 + 1.0;

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to +1.0, defaults to 1.0.

Outputs

Output
-1.0 to +1.0

Resources

Ticks: 22

Code: 22 words

Data: 2 words

Implementation

V27

See Also

`line_in.dsp`, `multiply.dsp`, `svfilter.dsp`

deemphcd.dsp

CD de-emphasis filter.

Description

This filter is a standard "feed-forward, feed-back" filter designed by Ayabe-san of MEI for CD de-emphasis.

Inputs

Input
Signal to process.

Outputs

Output
Processed signal.

Resources

Ticks: 17

Code: 17 words

Data: 7 words

Implementation

V20

delay4.dsp

Delays an input signal by up to 4 frames.

Description

Delays an input signal by up to 4 frames. This can be used to implement a FIR or IIR filter by connecting the output array to the inputs of a mixer.

The final output of this input can be connected to the input of another `delay4.dsp` to make a chain of arbitrary length. For very long delays you should use `delay_f1.dsp` which uses main memory instead of the more precious DSP data mamory.

Inputs

Input
Signal to delay.

Outputs

Output - 4 parts
Output part 0 is delayed one frame from the Input; Output part 1, two frames; and so on.

Resources

Ticks: 10

Code: 10 words

Data: 4 words

Implementation

V28

See Also

Mixer, `delay_f1.dsp`

delay_f1.dsp

Sends mono input to a delay line.

Description

This instrument writes input to an output FIFO. It is used as a building block for reverberation and echo effects when used in conjunction with a sample player (e.g. `sampler_16_f1.dsp`). It can only write to special Samples created using `CreateDelayLine()`.

Inputs

Input

Mono signal to delay.

Output FIFOs

OutFIFO

Writes monophonic, 16-bit sample data to delay line Sample attached to this port.

Resources

Ticks: 4

Code: 4 words

FIFOs: 1

Implementation

V20

See Also`delay_f2.dsp`, `sampler_16_f1.dsp`, `CreateDelayLine()`

delay_f2.dsp

Sends stereo input to a delay line.

Description

This instrument writes input to an output FIFO. It is used as a building block for reverberation and echo effects when used in conjunction with a sample player (e.g. `sampler_16_f2.dsp`). It can only write to special `Samples` created using `CreateDelayLine()`.

Inputs

Input - 2 parts
Stereo signal to delay.

Output FIFOs

OutFIFO
Writes stereo, 16-bit sample data to delay line `Sample` attached to this port.

Resources

Ticks: 12

Code: 10 words

FIFOs: 1

Implementation

V24

See Also

`delay_f1.dsp`, `sampler_16_f2.dsp`, `CreateDelayLine()`

depopper.dsp

Helps eliminate pops when switching sounds.

Description

This instrument can be used to reduce pops that can occur when stopping one sound and starting another. The sequence is as follows:

Connect instrument A to input of depopper.

Start A and depopper.

Before you want to stop A, set Gate on depopper to 0.0. This will latch the output of instrument A inside the depopper.

Stop A.

At this point you can switch to another instrument B and connect it to the depopper while holding A's old output.

Start B then set Gate on depopper to 1.0. The depopper will now smoothly crossfade from the held value from A to the current output of B.

Knobs

Gate - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

If greater than zero, signal passes, otherwise held.

Rate - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Amount to increment internal phase per execution frame. A higher number means it will crossfade from the held signal to the input signal more quickly. Range is 0.0 to 1.0. Defaults to 0.027, which results in a crossfade time of approximately 20ms.

Inputs

Input

Outputs

Output

Input signal or crossfaded output.

Resources

Ticks: 18

Code: 20 words

Data: 5 words

Implementation

V30

See Also

`envelope.dsp` `latch.dsp`

filter_1o1z.dsp

First Order, One Zero filter.

Description

First Order, One Zero filter using the following formula:

$$y(n) = A0*x(n) + A1*x(n-1)$$

where y(n) is Output, x(n) is Input and x(n-1) is Input at the prior sample tick.

Setting A1 positive gives a low-pass response; setting A1 negative gives a high-pass response. The bandwidth of this filter is fairly high, so it often serves a building block by being cascaded with other filters.

If A0 and A1 are both 0.5, then this filter is a simple averaging lowpass filter, with a zero at $SR/2 = 22050$ Hz.

If A0 is 0.5 and A1 is -0.5, then this filter is a high pass filter, with a zero at 0.0 Hz.

A thorough description of the digital filter theory needed to fully describe this filter is beyond the scope of this document.

Calculating coefficients is non-intuitive; the interested user is referred to one of the standard texts on filter theory (e.g., Moore, "Elements of Computer Music", section 2.4).

Knobs

A0, A1 - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Filter coefficients, -1.0 to 1.0, default = 0.5.

Inputs

Input
Signal to process.

Outputs

Output
Filtered signal.

Resources

Ticks: 11

Code: 11 words

Data: 4 words

Implementation

V29

See Also

svfilter.dsp

filter_3d.dsp

Sound spatialization filter.

Description

Used in the 3DSound code in the music library to simulate the head-related transfer function (HRTF) normally used in three-dimensional sound spatialization. The filter is a combination lowpass and notch, with the equation:

$$y(n) = \text{Feed} * x(n) + \text{Beta} * x(n-3) + \text{Alpha} * y(n-1)$$

where $y(n)$ is output, $y(n-1)$ is the previous output, $x(n)$ is the current input and $x(n-3)$ is the input three iterations ago. Alpha and Beta can be used to control a "lowpass" and "notch" characteristic somewhat independently. Feed should be set to $(1.0 - \text{Alpha} - \text{Beta})$, and controls the amount of the original input to mix into the output. The notch filter is centered at $(\text{sample rate})/4$, about 11 KHz.

When used with the 3DSound code, two such filters are used: one for each ear of the observer. The filter coefficients are set based on distance and angle of the observer to the sound. The lowpass character is maximised for sounds on the opposite side of the head from the ear or directly behind the observer, and is increased with distance. The notch character is maximised as sounds move directly in front or behind the observer.

Knobs

Alpha - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Lowpass filter coefficient, -1.0 to 1.0.

Beta - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Notch filter coefficient, -1.0 to 1.0.

Feed - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
Feed-forward coefficient, should normally be set to $(1.0 - \text{Alpha} - \text{Beta})$ for a stable filter.

Inputs

Input

Outputs

Output
Filter output.

Resources

Ticks: 18

Code: 18 words

Data: 7 words

Implementation

V30

See Also

`svfilter.dsp`, `filter_1olz.dsp`, `Create3DSound()`

svfilter.dsp

State-variable digital filter.

Description

State-variable digital filter.

To convert a frequency in Hz to the value needed to set the filter, the following formula, from Hal Chamberlain's "Musical Applications of MicroProcessors," is used:

$$F1 = 2 * \sin (\text{PI} * \text{CriticalFrequency} / \text{FrameRate})$$

where:

F1 is the frequency control value.

CriticalFrequency is the desired critical frequency in Hertz.

FrameRate is audio frame rate (e.g. 44100.0).

If CriticalFrequency / SampleRate is small, for example less than 1/5, you can use the approximation:

$$F1 = 2 * \text{PI} * \text{CriticalFrequency} / \text{FrameRate}$$

In both above cases, F1 is a floating point value suitable for passing to `SetKnob()`.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

F1 (as defined above) in the range of 0.0 to 1.0. For FrameRate of 44100 Hz, this corresponds to a critical frequency range of 0 to 7350 Hz. Remember that the relationship between critical frequency and F1 isn't linear. Defaults to 0.25 (approx 1759 Hz).

Resonance - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

1 / Q in the range of 0.0 to 1.0. Defaults to 0.125.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Amplitude of Output in the range of 0.0 to 1.0. Defaults to 1.0.

Inputs

Input

Outputs

Output

Low-pass output scaled by Amplitude.

LowPass, BandPass, HighPass

Low-pass, band-pass, and high-pass outputs of the filter. These are NOT scaled by Amplitude.

Resources

Ticks: 19

Code: 19 words

Data: 7 words

Implementation

V20

Caveats

The filter can be driven into oscillation to produce a sine wave. Because the frequency control is non-linear, however, it can't be used for such things as FM synthesis.

See Also`filter_1olz.dsp`

line_in.dsp

Taps stereo audio line in.

Description

Outputs the stereo Audio Input signal.

Each task that intends to use this instrument must successfully enable audio input `EnableAudioInput()` before the instrument can be loaded.

Any number of instances of `line_in.dsp` can be used without interaction.

Outputs

Output - 2 parts
Stereo line in signal.

Resources

Ticks: 6

Code: 6 words

Data: 2 words

Implementation

V23

Notes

A special license may be required to use audio input in a title. (!!! is this still true?)

See Also

`EnableAudioInput()`, `line_out.dsp`

line_out.dsp

Adds to stereo signal to send to audio line out.

Description

Mixes input stereo signal with the output of other `line_out.dsp` or `Mixer` instruments with `AF_F_MIXER_WITH_LINE_OUT` set. The result is sent to the DAC at the end of each audio frame.

The audio folio specially handles half-rate output instruments such as this. Their output is accumulated separately from full-rate output instruments and interpolated up to 44100 Hz to improve their fidelity.

Inputs

Input - 2 parts

Stereo signal to send to line out. You may use `AF_PART_LEFT` and `AF_PART_RIGHT` to address each part of the stereo signal.

Resources

Ticks: 8

Code: 8 words

Implementation

V21

See Also

`line_in.dsp`, `Mixer`

tapoutput.dsp

Taps accumulated stereo line out signal.

Description

This instrument permits reading the accumulated stereo output from all currently running output instruments (e.g., `line_out.dsp`, Mixer with `AF_F_MIXER_WITH_LINE_OUT` set). These signals are what is eventually sent to the audio DACs.

This can be used to snoop the audio output of an application without the application being aware of it (real handy for debugging!).

It can also be used to provide instrumentation, and such, for the accumulated audio output of the DSP (see examples).

Outputs

Output - 2 parts
Accumulated stereo output signal.

Resources

Ticks: 6

Code: 6 words

Data: 2 words

Implementation

V24

Examples

Combine with `delay_f2.dsp` to capture the output of an application to memory without the cooperation of the application (kind of like a screen grabber for audio).

Combine with a pair of `envfollower.dsp` instruments and a pair of Probes to make stereo VU meters for the accumulated audio of all applications currently running.

Combine with a pair of `envfollower.dsp` instruments, a pair of `maximum.dsp` instruments, and a pair of Probes to make a stereo peak level detector for the accumulated audio of all applications currently running.

Notes

You should only use this instrument when there's no direct way to get the signals you want. For example, if you want to add metering or peak detection to your own application (as opposed to every application that is running on the 3DO at a given time), split your audio (e.g. the output of a sub-mixer), between `line_out.dsp` and `envfollowers` instead of using `tapoutput.dsp`.

This instrument must be allocated at a lower priority than all the output instruments that you want to snoop. If not, you'll only get the accumulated results of all of the output instruments at higher (and perhaps equal) priority to `tapoutput.dsp`. Allocate at a priority of 0 to get the accumulated results of most output instruments.

See Also

`line_out.dsp`, `delay_f1.dsp`, `envfollower.dsp`, `maximum.dsp`, `capture_audio`, `minmax_audio`

sampler_16_f1.dsp

Fixed-rate mono 16-bit sample player.

Description

This instrument plays a monophonic 16-bit sample at the instrument's execution rate (e.g., 44100 samples/sec).

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
0.0 to 1.0. Defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 6

Code: 6 words

Data: 2 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_f2.dsp, sampler_16_v1.dsp, sampler_raw_f1.dsp

sampler_16_f2.dsp

Fixed-rate stereo 16-bit sample player.

Description

This instrument plays a stereophonic 16-bit sample at the instrument's execution rate (e.g., 44100 samples/sec).

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
0.0 to 1.0. Defaults to 1.0.

Outputs

Output - 2 parts
Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 17

Code: 14 words

Data: 3 words

FIFOs: 1

Implementation

V27

See Also

`sampler_16_f1.dsp`, `sampler_16_v2.dsp`

sampler_16_v1.dsp

Variable-rate mono 16-bit sample player.

Description

Variable-rate monophonic 16-bit sample player.

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 24

Code: 5 words per instrument + 15 words shared overhead

Data: 4 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_f1.dsp, sampler_16_v2.dsp

sampler_16_v2.dsp

Variable-rate stereo 16-bit sample player.

Description

Variable-rate stereo 16-bit sample player.

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output - 2 parts

Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 47

Code: 46 words

Data: 9 words

FIFOs: 1

Implementation

V27

See Also

sampler_cbd2_v2.dsp, sampler_16_v1.dsp

sampler_8_f1.dsp

Fixed-rate mono 8-bit sample player.

Description

This instrument plays a monophonic 8-bit sample at the instrument's execution rate (e.g., 44100 samples/sec).

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
0.0 to 1.0. Defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 6

Code: 6 words

Data: 2 words

FIFOs: 1

Implementation

V27

Caveats

8-bit audio sounds inherently BAD because there is very little information in each sample. Use one of the compressed data formats such as SQS2 for mono samples, or CBD2 for stereo samples, for better quality. The only reason to use 8-bit audio is if you don't have the 16-bit original recording.

See Also

sampler_8_f2.dsp, sampler_sqs2_f1.dsp

sampler_8_f2.dsp

Fixed-rate stereo 8-bit sample player.

Description

This instrument plays a stereophonic 8-bit sample at the instrument's execution rate (e.g., 44100 samples/sec).

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
0.0 to 1.0. Defaults to 1.0.

Outputs

Output - 2 parts
Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 17

Code: 14 words

Data: 3 words

FIFOs: 1

Implementation

V27

Caveats

8-bit audio sounds inherently BAD because there is very little information in each sample. Use one of the compressed data formats such as SQS2 for mono samples, or CBD2 for stereo samples, for better quality. The only reason to use 8-bit audio is if you don't have the 16-bit original recording.

See Also

sampler_8_f1.dsp, sampler_cbd2_f2.dsp

sampler_8_v1.dsp

Variable-rate mono 8-bit sample player.

Description

Variable-rate monophonic 8-bit sample player.

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 24

Code: 5 words per instrument + 15 words shared overhead

Data: 4 words

FIFOs: 1

Implementation

V27

Caveats

8-bit audio sounds inherently BAD because there is very little information in each sample. Use one of the compressed data formats such as SQS2 for mono samples, or CBD2 for stereo samples, for better quality. The only reason to use 8-bit audio is if you don't have the 16-bit original recording.

See Also

sampler_8_v2.dsp, sampler_sqs2_v1.dsp

sampler_8_v2.dsp

Variable-rate stereo 8-bit sample player.

Description

Variable-rate stereophonic 8-bit sample player.

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output - 2 parts

Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 47

Code: 46 words

Data: 9 words

FIFOs: 1

Implementation

V27

Caveats

8-bit audio sounds inherently BAD because there is very little information in each sample. Use one of the compressed data formats such as SQS2 for mono samples, or CBD2 for stereo samples, for better quality. The only reason to use 8-bit audio is if you don't have the 16-bit original recording.

See Also

sampler_8_v1.dsp, sampler_cbd2_v2.dsp

sampler_adp4_v1.dsp

Variable-rate mono sample player with ADPCM Intel/DVI 4:1 decompression.

Description

This instrument plays a sample, at variable rate, that has been previously compressed to a 4-bit Intel/DVI format. Use SoundHack or SquashSnd to compress the sample.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 44100.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 106

Code: 9 words per instrument + 99 words shared overhead

Data: 11 words per instrument + 98 words shared overhead

FIFOs: 1

Implementation

V24

Caveats

Unlike most other sample player instruments this one doesn't support playing the sample above the system sample rate.

See Also

sampler_16_v1.dsp, sampler_cbd2_v1.dsp, sampler_sqs2_v1.dsp

sampler_cbd2_f1.dsp

Fixed-rate mono sample player with CBD2 2:1 decompression.

Description

Fixed-rate mono CBD2-format sample player.

This instrument plays a sample, at a fixed rate, that has been previously compressed to the 8-bit CBD2 format. Run SoundHack or SquashSnd to compress the sample.

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
0..1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 45

Code: 42 words

Data: 7 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_f1.dsp, sampler_adp4_v1.dsp, sampler_cbd2_f2.dsp,
sampler_sqs2_f1.dsp

sampler_cbd2_f2.dsp

Fixed-rate stereo sample player with CBD2 2:1 decompression.

Description

Fixed-rate stereo CBD2-format sample player.

This instrument plays a sample, at a fixed rate, that has been previously compressed to the 8-bit CBD2 format. Run SoundHack or SquashSnd to compress the sample.

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to 1.0, defaults to 1.0.

Outputs

Output - 2 parts
Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 69

Code: 48 words

Data: 6 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_f2.dsp, sampler_cbd2_f1.dsp

sampler_cbd2_v1.dsp

Variable-rate mono sample player with CBD2 2:1 decompression.

Description

Fixed-rate mono CBD2-format sample player.

This instrument plays a sample, at variable rate, that has been previously compressed to the 8-bit CBD2 format. Run SoundHack or SquashSnd to compress the sample.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE
SampleRate in Hertz. Range is 0.0 to 44100.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 63

Code: 57 words

Data: 10 words

FIFOs: 1

Implementation

V27

Caveats

Unlike most other sample player instruments this one doesn't support playing the sample above the system sample rate.

See Also

sampler_16_v1.dsp, sampler_adp4_v1.dsp, sampler_cbd2_v2.dsp,
sampler_sqs2_v1.dsp

sampler_cbd2_v2.dsp

Variable-rate stereo sample player with CBD2 2:1 decompression.

Description

Variable-rate stereo CBD2-format sample player.

This instrument plays a sample, at variable rate, that has been previously compressed to the 8-bit CBD2 format. Run SoundHack or SquashSnd to compress the sample.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 44100.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output - 2 parts

Stereo output.

Input FIFOs

InFIFO

Resources

Ticks: 91

Code: 66 words

Data: 9 words

FIFOs: 1

Implementation

V27

Caveats

Unlike most other sample player instruments this one doesn't support playing the sample above the system sample rate.

See Also

sampler_16_v2.dsp, sampler_cbd2_v1.dsp

sampler_drift_v1.dsp

Variable-rate mono 16-bit sample player with drift output (suitable for flanging).

Description

Variable-rate mono 16-bit sample player with drift tracking. This instrument is normally used to implement variable-rate delays used for flange, chorus, and spatialization effects.

The Drift output can be fed back to the SampleRate knob as negative feedback which forces the delay to a given offset.

$\text{SampleRate} = 44100.0 - \text{FeedbackScalar} * (\text{Drift} - \text{DesiredOffset})$

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE
SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Drift

When interpreted as AUDIO_SIGNAL_TYPE_WHOLE_NUMBER, this is the number of samples consumed minus the number of frames. If played at a SampleRate of 44100.0, the drift should stay 0. Drift is reset to 0 whenever the instrument is restarted. Signed value in the range of -32768.0 to 32767.0 when interpreted as AUDIO_SIGNAL_TYPE_WHOLE_NUMBER.

Input FIFOs

InFIFO

Resources

Ticks: 28

Code: 9 words per instrument + 15 words shared overhead

Data: 5 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_v1.dsp, delay_f1.dsp, timesplus_noclip.dsp

sampler_raw_f1.dsp

Fixed-rate mono 16-bit sample player without amplitude scaling.

Description

This instrument plays a monophonic 16-bit sample at the instrument's execution rate (e.g., 44100 samples/sec). It does no scaling by "Amplitude" knob, so the sample data is played unmodified.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 4

Code: 4 words

Data: 1 word

FIFOs: 1

Implementation

V27

See Also

sampler_16_f1.dsp

sampler_sqs2_f1.dsp

Fixed-rate mono sample player with SQS2 2:1 decompression.

Description

Fixed-rate SQS2-format sample player.

This instrument plays a monophonic SQS2 compressed sample at the system sample rate, normally 44100 Hz. The sample can be compressed using the SquashSnd tool.

The M2 system does this decompression in hardware, so it has very little DSP overhead.

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to 1.0. Defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 6

Code: 6 words

Data: 2 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_f1.dsp, sampler_adp4_v1.dsp, sampler_cbd2_f1.dsp

sampler_sqs2_v1.dsp

Variable-rate mono sample player with SQS2 2:1 decompression.

Description

Variable-rate SQS2-format sample player.

This instrument is limited to a pitch one octave above base pitch, if recorded at 44.1 kHz. If it is recorded at 22 kHz, the pitch can go two octaves up.

The M2 system does this decompression in hardware, so it has very little DSP overhead.

Knobs

SampleRate - AUDIO_SIGNAL_TYPE_SAMPLE_RATE

SampleRate in Hertz. Range is 0.0 to 88200.0. Default is 44100.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to 1.0, defaults to 1.0.

Outputs

Output

Input FIFOs

InFIFO

Resources

Ticks: 24

Code: 5 words per instrument + 15 words shared overhead

Data: 4 words

FIFOs: 1

Implementation

V27

See Also

sampler_16_v1.dsp, sampler_adp4_v1.dsp, sampler_cbd2_v1.dsp

chaos_1d.dsp

One-dimensional chaotic function generator (the logistic map).

Description

This instrument generates numbers by iterating the logistic map equation:

$$x0 = 4 * \text{Scalar} * x1(1.0 - x1), 0.0 \leq x0 \leq 1.0$$

where x1 is the output from last iteration, and x0 the current one.

At low values of Scalar, the output is fairly constant. As Scalar is increased, the output rises, until at about Scalar = 0.75 the output begins to oscillate in a two-limit cycle. As Scalar is increased further, the output bifurcates again into a four-limit cycle, then into longer cycles and eventually into a pseudo-random or chaotic sequence, with regions of stability or near-stability.

The output is available in two forms, as a ramp between successive values of x, and as a step function.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0. Controls how frequently x0 is calculated.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0. Scales the value of x0.

Scalar - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

0.0 to +1.0, defaults to 0.825.

Outputs

Output

0.0 to +1.0. Outputs the latest value of x0.

InterpolatedOutput

0.0 to +1.0. Interpolates between the current and previous values of x0.

Resources

Ticks: 40

Code: 36 words

Data: 8 words

Implementation

V30

See Also

noise.dsp, rednoise.dsp

impulse.dsp

Impulse waveform generator.

Description

This instrument is a simple impulse generator. The output is either zero or the Amplitude value. The width of the impulse is one sample. This is useful for pinging filters.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 14

Code: 12 words

Data: 4 words

Implementation

V20

See Also

`pulse.dsp`, `square.dsp`

noise.dsp

White noise generator.

Description

This instrument uses a pseudo-random number generator to produce white noise. A new random number is generated every frame.

Knobs

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED
-1.0 to 1.0, defaults to 1.0.

Outputs

Output
-1.0 to 1.0.

Resources

Ticks: 6

Code: 6 words

Data: 2 words

Implementation

V20

See Also

`rednoise.dsp`, `randomhold.dsp`

pulse.dsp

Pulse wave generator.

Description

Pulse wave generator with variable pulse width.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

PulseWidth - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

Level for comparator used to generate pulse wave. 0 gives a 50% duty cycle pulse wave.

Positive values cause the positive portion of the pulse to be narrower than the negative.

Negative values cause the negative portion of the pulse to be narrower than the positive.

The range is -1.0 to +1.0, defaults to 0.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 16

Code: 14 words

Data: 5 words

Implementation

V21

See Also

`square.dsp`, `impulse.dsp`, `pulse_lfo.dsp`

rednoise.dsp

Red noise generator.

Description

This instrument interpolates straight line segments between pseudo-random numbers to produce "red" noise. It is a grittier alternative to the white generator `noise.dsp`. It is also useful as a slowly changing random control generator for natural sounds.

Knobs

Frequency - `AUDIO_SIGNAL_TYPE_OSC_FREQ`

Frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0. Controls how frequently a new random value is chosen.

Amplitude - `AUDIO_SIGNAL_TYPE_GENERIC_SIGNED`

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 24

Code: 21 words

Data: 6 words

Implementation

V20

See Also

`noise.dsp`, `randomhold.dsp`, `rednoise_lfo.dsp`

sawtooth.dsp

Sawtooth wave generator.

Description

Sawtooth wave generator.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 8

Code: 8 words

Data: 4 words

Implementation

V20

See Also

pulse.dsp, square.dsp, triangle.dsp

square.dsp

Square wave generator.

Description

Square wave generator. It has a woody, clarinet-like sound.

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 14

Code: 12 words

Data: 4 words

Implementation

V21

See Also

`pulse.dsp`, `sawtooth.dsp`, `triangle.dsp`, `square_lfo.dsp`

triangle.dsp

Triangle wave generator.

Description

Triangle wave generator. The waves it produces looks like: /\ /\ .

Knobs

Frequency - AUDIO_SIGNAL_TYPE_OSC_FREQ

Oscillator frequency in Hertz. Range is -22050.0 to +22050.0. Default is 440.0.

Amplitude - AUDIO_SIGNAL_TYPE_GENERIC_SIGNED

-1.0 to +1.0, defaults to 1.0.

Outputs

Output

-1.0 to +1.0

Resources

Ticks: 17

Code: 15 words

Data: 4 words

Implementation

V20

See Also

pulse.dsp, sawtooth.dsp, square.dsp, triangle_lfo.dsp

Chapter 6

Music Link Library Calls

This section presents the reference documentation for the Music link library.

--ATAG-File-Format--

Simple audio object file format for Samples, Envelopes, Delay lines, and Tunings.

File Format

```
{
    // header
    PackedID 'ATAG'
    uint32    size of AudioTagHeader
    AudioTagHeader

    // data
    PackedID 'BODY'
    uint32    size of body data in bytes. Use 0 size when no body
data
    is present
    uint8 data [size]
}
```

Description

This file format provides a low-overhead method of loading data-oriented audio Items from disc. It supports the following types of audio folio Items:

- Sample
- Delay Line
- Envelope
- Tuning

The format is IFF-like in that it has chunks, but is not actually an IFF FORM. This is because this format is meant to be as simple to parse as possible.

Because of its IFF-like chunk nature, such a file can be embedded directly into an IFF FORM and parsed using the IFF folio. `LoadATAG()` can't be used to deal with extracting this from an IFF FORM. Instead call `ValidateAudioTagHeader()` and `CreateItem()` after extracting the chunks.

Support for parsing ATAG files is provided in `libmusic.a`.

Associated Files

<:audio:atag.h>, `libmusic.a`

Examples

Here is an example his function which demonstrates how to create an Item based on ATAG and BODY chunk contents:

```
CreateATAGItem (const AudioTagHeader *atag, void *body)
{
    return CreateItemVA (MKNODEID (AUDIONODE, atag->athd_NodeType),
                        body ? AF_TAG_ADDRESS      : TAG_NOP, body,
```

```
        body ? AF_TAG_AUTO_FREE_DATA : TAG_NOP, TRUE,  
        TAG_JUMP, atag->athd_Tags);  
}
```

Since this uses AF_TAG_AUTO_FREE_DATA to hand off the body data to the Item, the body data must be allocated with MEMTYPE_TRACKSIZE. Both AF_TAG_ADDRESS and AF_TAG_AUTO_FREE_DATA tags are illegal for delay lines, so don't specify these tags when there's no body data (a necessary condition for a delay line).

Note that this function as written expects the caller to have validated the chunk contents.

See Also

LoadATAG(), ValidateAudioTagHeader(), Sample, Envelope, Tuning

AudioTagHeaderSize

Returns size of AudioTagHeader for given number of tags.

Synopsis

```
int32 AudioTagHeaderSize (int32 numTags)
```

Description

This function returns the size of an AudioTagHeader in bytes for the given number of tags.

Arguments

numTags
Number of tags.

Return Value

Size of the AudioTagHeader in bytes.

Implementation

Macro function implemented in `<:audio:atag.h>` V29.

Associated Files

`<:audio:atag.h>`

See Also

`--ATAG-File-Format--`, `ValidateAudioTagHeader()`

LoadATAG

Load an ATAG simple audio object format file.

Synopsis

```
Item LoadATAG (const char *fileName)
```

Description

This function loads an ATAG simple audio object format file. This format is a low overhead file format which can contain samples, envelopes, delay line descriptions, and tunings.

The result of a successful load is an audio folio Item of the type stored in the file. If the Item has some data (e.g., sample data, envelope table, tuning table), then the Item is created with { AF_TAG_AUTO_FREE_DATA, TRUE } to facilitate simple and complete deletion of the Item. The returned Item is given the same name as the ATAG file.

Use DeleteItem() to dispose of the resulting Item when done with it.

Arguments

fileName
Name of an ATAG file to load.

Return Value

The procedure returns an Audio Item of the type stored in the file (a positive value) if successful, or an error code (a negative value) if an error occurs.

Implementation

Library function implemented in libmusic.a V29.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:atag.h>, libmusic.a, System.m2/Modules/audio

See Also

```
--ATAG-File-Format--, ValidateAudioTagHeader(), Sample, Tuning, Envelope,  
LoadSample(), CreateItem(), DeleteItem()
```

ValidateAudioTagHeader

Validate the contents of an AudioTagHeader (ATAG chunk).

Synopsis

```
Err ValidateAudioTagHeader (const AudioTagHeader *atag, uint32  
    atagSize)
```

Description

This function validates the contents of an AudioTagHeader read from an ATAG chunk. Checks the following things:

- Whether atagSize is large enough for complete AudioTagHeader and all of the tags specified in athd_NumTags.
- That there is at least 1 tag.
- The tag list contains no invalid tags: TAG_JUMP, TAG_END, AF_TAG_ADDRESS, AF_TAG_AUTO_FREE_DATA.
- The tag list is terminated by a TAG_END.

Arguments

atag
 Pointer to AudioTagHeader (ATAG chunk)

atagSize
 Size of the ATAG chunk in bytes.

Return Value

The procedure returns a non-negative value on success, or a negative error code on failure.

Implementation

Library function implemented in libmusic.a V29.

Associated Files

<:audio:atag.h>, libmusic.a

See Also

--ATAG-File-Format--, LoadATAG()

CollectionClass

Multiple parallel sequences and collections.

Description

Objects of this class can maintain and playback in parallel an assembly of sequences and other collections.

Super Class

JuggleeClass

Methods

!!!

Implementation

V20

Associated Files

<:audio:juggler.h>, libmusic.a

See Also

SequenceClass

JuggleeClass

Root juggler class.

Description

This is the root class used to derive other juggler classes. It cannot be used as a stand-alone class.

Super Class

None

Methods

!!!

Implementation

V20

Associated Files

<:audio:juggler.h>, libmusic.a

See Also

CollectionClass, SequenceClass

SequenceClass

A single sequence of events.

Description

Objects of this class can maintain and playback a single sequence of events.

Super Class

JuggleeClass

Methods

!!!

Implementation

V20

Associated Files

<:audio:juggler.h>, libmusic.a

See Also

CollectionClass

AbortObject

Abnormally stops a juggler object.

Synopsis

```
int32 AbortObject( CObject *obj, Time stopTime )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro abnormally stops an object at the specified time (measured in audio clock ticks). Once an object is stopped, it's removed from the active object list, and will no longer be played by the juggler when the juggler is bumped.

!!! how does this differ from `StopObject()`.
. no completion message sent for `AbortObject()` ?

Arguments

`obj`
A pointer to the `CObject` data structure for the object.

`stopTime`
A value indicating the time to stop the object.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`StopObject()`, `SetObjectInfo()`, `GetObjectInfo()`, `StartObject()`,
`AllocObject()`, `FreeObject()`, `GetNthFromObject()`, `RemoveNthFromObject()`,
`PrintObject()`

AllocObject

Allocates memory for an object.

Synopsis

```
Err AllocObject( CObject *obj, int32 n )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro allocates memory for an object. It checks the amount of RAM required for the elements of the specified object, and then uses the argument *n* to see how many of those elements exist and need memory allocation. This procedure is typically used for a sequence object, in which case *n* specifies how many events must be stored in the sequence.

Arguments

obj

A pointer to the CObject data structure for the object.

n

The number of elements that must be stored in memory.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`AllocObject()`, `FreeObject()`, `GetNthFromObject()`, `GetObjectInfo()`,
`PrintObject()`, `RemoveNthFromObject()`, `SetObjectInfo()`, `StartObject()`,
`StopObject()`

BumpJuggler

Bumps the juggler data-structure index.

Synopsis

```
int32 BumpJuggler( Time CurrentTime, Time *NextTime,  
                  int32 CurrentSignals, int32 *NextSignals )
```

Description

This procedure provides the juggler with a current time value. The juggler checks unexecuted events in any currently active sequences or jobs to see if their time value is less than or equal to the current time value. If so, it executes the event. The juggler also checks all unexecuted events in all currently active sequences or jobs to see which event should be executed next. It writes the time of that event into the NextTime variable so the calling task can wait until that time to bump the juggler once again. The time value supplied is an arbitrary value in ticks and must be supplied from an external source. Typically, the time value comes from the audio clock.

At this writing, BumpJuggler() does not work with signals, so ignore this paragraph.This procedure may also work with signals specified for each event. The current signal mask lists current signal bits. The juggler checks unexecuted events for events associated with current signal bits, and executes those events. It writes the next set of signals to wait for in the NextSignals signal mask. To use this feature properly, CurrentSignals should be set to 0 the first time you call BumpJuggler(). For subsequent calls, you would have your program wait for the signals to be set (with WaitSignal()), or you would wait until the NextTime time is reached. When either the signal arrives or the time occurs, you would call BumpJuggler() again with the new time and any signals that have arrived in the wait interval.

See the example program "playmf.c" to see how BumpJuggler() is used.

Arguments

CurrentTime

Value indicating the current time that the procedure is called.

NextTime

Pointer to a value where this procedure writes the time that it should be called again.

CurrentSignals

The current signal mask.

NextSignals

Pointer to a signal mask where this procedure writes the signal bits for which the task should wait before making this call again.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs. If everything is finished or there are no active objects, this procedure returns 1.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:juggler.h>, libmusic.a

See Also

InitJuggler(), TermJuggler()

CreateObject

Creates an object of the given class.

Synopsis

```
COBObject *CreateObject( COBClass *Class )
```

Description

This procedure is part of the juggler object-oriented toolbox that supports the various classes of juggler objects. This procedure creates an object of the given class. Execute the object's `Init()` method to correctly set up the object.

Arguments

Class

A pointer to the COBClass data structure for the class (currently available: `&SequenceClass` and `&CollectionClass` as defined in `<:audio:juggler.h>`).

Return Value

This procedure returns a pointer to the COBObject data structure defining the object if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V20`.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`DefineClass()`, `DestroyObject()`, `ValidateObject()`

DefineClass

Defines a class of objects.

Synopsis

```
int32 DefineClass( COBClass *Class, COBClass *SuperClass,  
                  int32 DataSize )
```

Description

At this writing, this call is for internal use only, and not available to user tasks.

This procedure is part of the juggler object-oriented toolbox that supports the various classes of juggler objects. This procedure defines a new class of objects with the given size. All methods for this class are inherited from the specified SuperClass. New methods are added to the class by setting function pointers in the Class structure.

See the chapter "Playing Juggler Objects" in the Portfolio Programmer's Guide for more information about object-oriented programming and the data structures necessary for this procedure.

Arguments**Class**

A pointer to the COBClass data structure for the class.

SuperClass

A pointer to the COBClass data structure for the superclass.

DataSize

A value indicating the size of the new class.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:cobj.h>, libmusic.a

See Also

CreateObject(), DestroyObject(), ValidateObject()

DestroyObject

Destroys an object.

Synopsis

```
int32 DestroyObject( COBObject *Object )
```

Description

This procedure is part of the juggler object-oriented toolbox that supports the various classes of juggler objects. This procedure gets rid of an object. Execute the object's `Term()` method to close the object, then use this call to get rid of the object by deleting its `COBObject` data structure and freeing the memory used to store it.

Arguments

Object

A pointer to the `COBObject` data structure for the object.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V20`.

Associated Files

<:audio:cobj.h>, `libmusic.a`

See Also

`CreateObject()`, `DefineClass()`, `ValidateObject()`

FreeObject

Frees memory allocated for an object.

Synopsis

```
Err FreeObject( CObject *obj )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro frees memory allocated for the object using `AllocObject()`.

Arguments

`obj`
A pointer to the `CObject` data structure for the object.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `GetObjectInfo()`, `StopObject()`, `AllocObject()`,
`GetNthFromObject()`, `RemoveNthFromObject()`, `PrintObject()`, `StartObject()`

GetNthFromObject

Gets the nth element of a collection.

Synopsis

```
Err GetNthFromObject( CObject *obj, int32 n,  
                     (void**) ptr )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro returns a pointer to the nth element of a collection so that a task can find out what elements are included in the collection. Note that element numbering within a collection starts at 0.

Arguments

- obj
A pointer to the CObject data structure for the object.
- n
A value indicating the element to get, starting at 0.
- ptr
A variable in which to store a pointer to the element found.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `GetObjectInfo()`, `StopObject()`, `AllocObject()`,
`FreeObject()`, `RemoveNthFromObject()`, `PrintObject()`, `StartObject()`

GetObjectInfo

Gets the current settings of an object.

Synopsis

```
Err GetObjectInfo( CObject *obj, TagArg *tags )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros. This method macro replaces the TagArg values in the specified tag arg list with the values currently defined for the object. Use this procedure to get information about the current state of an object.

See `SetObjectInfo()` for a list of the tag args supported.

Arguments

`obj`
A pointer to the CObject data structure for the object.

`tags`
A pointer to the TagArg list.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `StopObject()`, `AllocObject()`, `FreeObject()`,
`GetNthFromObject()`, `RemoveNthFromObject()`, `PrintObject()`, `StartObject()`

InitJuggler

Initializes the juggler mechanism for controlling events.

Synopsis

```
int32 InitJuggler( void )
```

Description

This procedure initializes the juggler scheduling system for controlling arbitrary events. Note that the juggler must be initialized before calling any object-oriented, juggler, or MIDI playback procedures.

See the chapter "Playing Juggler Objects" in the Portfolio Programmer's Guide for more information about the juggler and how it works.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:juggler.h>, libmusic.a, System.m2/Modules/audio

See Also

```
TermJuggler(), BumpJuggler()
```

PrintObject

Prints debugging information about an object.

Synopsis

```
Err PrintObject( CObject *obj )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer or with one of the defined method macros.

This method macro prints debugging information about an object. The information includes a pointer to the object.

Arguments

obj
A pointer to the CObject data structure for the object.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, libmusic.a

See Also

SetObjectInfo(), GetObjectInfo(), StartObject(), StopObject(),
AllocObject(), FreeObject(), GetNthFromObject(), RemoveNthFromObject()

RemoveNthFromObject

Removes the nth element of a collection.

Synopsis

```
Err RemoveNthFromObject( CObject *obj, int32 n )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer or with one of the defined method macros.

This method macro removes the reference to the nth element (a sequence or a collection) of a collection. Numbering starts from 0.

Note that using this procedure to remove a sequence or collection from a higher collection doesn't delete the sequence or collection, but merely removes its reference in the higher collection so that the element is no longer part of the collection.

Arguments

obj

A pointer to the CObject data structure for the object.

n

A value indicating the element to remove, starting at 0.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `GetObjectInfo()`, `StopObject()`, `AllocObject()`,
`FreeObject()`, `GetNthFromObject()`, `PrintObject()`, `StartObject()`

SetObjectInfo

Sets values in the object based on tag args.

Synopsis

```
Err SetObjectInfo( CObject *obj, TagArg *tags )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro sets the values in the object based on the supplied tag arguments. Valid tag arguments are as follows: Tags for Jugglee SuperClass

!!! the formatting of the following paragraph is totally hosed:

As the jugglee is the superclass for the other juggler classes, its methods are inherited by other juggler classes. The following tags are for SetObjectInfo() for the jugglee superclass: JGLR_TAG_CONTEXT <usercontext> User-specified context; this may be a pointer to almost anything. It is passed to the Interpreter function. JGLR_TAG_START_DELAY <ticks> The delay interval before starting this object. This is used to stagger the execution of parallel objects within a collection for canon. JGLR_TAG_REPEAT_DELAY <ticks> The time to wait between repetitions. JGLR_TAG_STOP_DELAY <ticks> The time to wait after stopping. JGLR_TAG_START_FUNCTION <*function(object, time)> This function is called before executing the first element of the object. JGLR_TAG_REPEAT_FUNCTION <*function(object, time)>, JGLR_TAG_STOP_FUNCTION <*function(object, time)>, JGLR_TAG_MUTE <flag> If the flag is true, the object will be muted when played. The Interpreter function must look at the mute flag. Tags for Sequence

A sequence contains an array of events that are to be executed over time. An event consists of a timestamp, followed by a user-defined, fixed-size data field. A sequence keeps track of time, advances through the events in its array, and calls a user-defined Interpreter function to "play" the event.

Sequences will typically contain arrays of MIDI messages from a MIDI file. Sequences may also contain other things; these other things each require a different Interpreter function. The possible sequence contents and their functions are as follows: FunctionPointers, Data schedule arbitrary functions. Cel, Corners draw this cel. Knob, Value tweak a knob for timbral sequences. Collections schedule scores at a very high level.

The following are tags for a sequence: JGLR_TAG_INTERPRETER_FUNCTION Specifies the user-defined function to be called for interpreting events in a sequence. This function is called at the time specified in the event. When the function is called, it is passed a pointer to one of your events. It is also passed a pointer to your context data structure that was set using JGLR_TAG_CONTEXT and SetObjectInfo(). The function prototype for interpreting events is given below: int32 InterpretEvent(Sequence *SeqPtr, void *CurrentEvent, void *UserContext) JGLR_TAG_MAX Sets the maximum number of events that can be used. JGLR_TAG_MANY Sets the actual number of events that can be used. JGLR_TAG_EVENTS <void *Events> Sets a pointer to your event array. JGLR_TAG_EVENT_SIZE <size_in_bytes> Sets the size of a single event in bytes so that an index into an array can be constructed. Other Tags

Other tag arguments are listed below:

JGLR_TAG_SELECTOR_FUNCTION

JGLR_TAG_AUTO_DELETE

JGLR_TAG_DURATION

Arguments

obj

A pointer to the CObject data structure for the object.

tags

A pointer to the TagArg list.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, libmusic.a

See Also

`GetObjectInfo()`, `StartObject()`, `StopObject()`, `AllocObject()`,
`FreeObject()`, `GetNthFromObject()`, `RemoveNthFromObject()`, `PrintObject()`

StartObject

Starts an object so the juggler will play it.

Synopsis

```
Err StartObject( CObject *obj, uint32 time, int32 nrep,  
                CObject *par )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer or with one of the defined method macros.

This method macro starts an object at the given time. Once an object is started, it's added to the active object list so that the juggler will play it when the juggler is bumped. The "nrep" argument sets an object to be played a set number of repetitions.

Arguments

- obj
A pointer to the CObject data structure for the object.
- time
A value indicating the time in audio clock ticks to start the object.
- nrep
A value indicating the number of occurrences of the object.
- par
A pointer to the CObject data structure that is the parent of this object.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `GetObjectInfo()`, `StopObject()`, `AllocObject()`,
`FreeObject()`, `GetNthFromObject()`, `RemoveNthFromObject()`, `PrintObject()`

StopObject

Stops an object so the juggler won't play it.

Synopsis

```
int32 StopObject( CObject *obj, uint32 time )
```

Description

This is a method macro that is part of the juggler object-oriented toolbox. An object's method can be called explicitly through the class structure function pointer, or with one of the defined method macros.

This method macro stops an object at the specified time (measured in audio clock ticks). Once an object is stopped, it's removed from the active object list, and will no longer be played by the juggler when the juggler is bumped.

Arguments

obj

A pointer to the CObject data structure for the object.

time

A value indicating the time to stop the object.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:cobj.h>` V20.

Associated Files

`<:audio:cobj.h>`, `libmusic.a`

See Also

`SetObjectInfo()`, `GetObjectInfo()`, `StartObject()`, `AllocObject()`,
`FreeObject()`, `GetNthFromObject()`, `RemoveNthFromObject()`, `PrintObject()`

TermJuggler

Terminates the juggler mechanism for controlling events.

Synopsis

```
int32 TermJuggler( void )
```

Description

This procedure terminates the juggler, which eliminates support for object-oriented calls, juggler playback, and MIDI score playback. A task should call this procedure when finished with the juggler.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:juggler.h>, libmusic.a

See Also

InitJuggler(), BumpJuggler()

ValidateObject

Validates an object.

Synopsis

```
int32 ValidateObject( COBObject *cob )
```

Description

This procedure is part of the juggler object-oriented toolbox that supports the various classes of juggler objects. This procedure returns an error code if the object is not a valid object that is, if the COBObject data structure element dedicated to object validity doesn't confirm validity.

Arguments

`cob`

A pointer to the COBObject data structure for the object.

Return Value

This procedure returns 0 if the object is valid or an error code (a negative value) if the object isn't valid.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:cobj.h>, libmusic.a

See Also

CreateObject(), DefineClass(), DestroyObject()

DeleteSampleInfo

Frees SampleInfo.

Synopsis

```
void DeleteSampleInfo (SampleInfo *smpi)
```

Description

Frees SampleInfo created by GetAIFFSampleInfo(). Frees data pointed to by smpi_Data if smpi_Flags contains the flag ML_SAMPLEINFO_F_SAMPLE_ALLOCATED.

Arguments

smpi
SampleInfo to free. Can be NULL.

Implementation

Library call implemented in libmusic.a V27.

Associated Files

<:audio:parse_aiff.h>, libmusic.a

See Also

GetAIFFSampleInfo()

DumpSampleInfo

Dumps SampleInfo structure returned by
GetAIFFSampleInfo().

Synopsis

```
void DumpSampleInfo (const SampleInfo *smpi, const char *banner)
```

Description

Dumps SampleInfo structure returned by GetAIFFSampleInfo() to debugging terminal.

Arguments

smpi

SampleInfo to dump.

banner

Optional banner to print. Can be NULL.

Implementation

Library call implemented in libmusic.a V30.

Associated Files

<:audio:parse_aiff.h>, libmusic.a

See Also

GetAIFFSampleInfo()

GetAIFFSampleInfo

Parses an AIFF sample file and return a SampleInfo data structure.

Synopsis

```
Err GetAIFFSampleInfo (SampleInfo **resultSampleInfo, const char
*fileName, uint32 flags)
```

Description

Parse an AIFF sample file on disk; allocate and fill out a SampleInfo structure with the relevant information.

The currently supported sample file formats are:

- AIFF

- AIFC

The SampleInfo contains a fairly complete description of the contents of the sample file. See `<:audio:parse_aiff.h>` for details.

You may interrogate the contents of the SampleInfo for your own uses. You may also pass it to `SampleInfoToTags()` to prepare a tag list to pass to `CreateSample()`.

When done with the SampleInfo, free it with `DeleteSampleInfo()`.

Arguments

resultSampleInfo

Pointer to a pointer to a SampleInfo structure that is allocated and filled out.

fileName

Name of an AIFF file to parse.

flags

Set of flags to control parsing.

The legal flags are:

ML_GETSAMPLEINFO_F_SKIP_DATA

When set causes `GetAIFFSampleInfo()` to get format information only. When not set, the sample data is loaded and a pointer to the loaded data is stored in `smpl_Data`.

Return Value

The procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V27`.

Associated Files

`<:audio:parse_aiff.h>`, `libmusic.a`, `libspmath.a`, `System.m2/Modules/iff`

See Also

SampleInfoToTags(), DeleteSampleInfo(), DumpSampleInfo(), LoadSample(),
Sample

LoadSample

Loads a Sample from an AIFF or AIFC file.

Synopsis

```
Item LoadSample (const char *fileName)
```

Description

This procedure allocates task memory and creates a sample item there that contains the digital-audio recording from the specified file. The file must be either an AIFF file or an AIFC file. These can be created using almost any sound development tool including AudioMedia, Alchemy, CSound, and SoundHack.

AIFC files contain compressed audio data. These can be created from an AIFF file using the SquashSound MPW tool from The 3DO Company.

Most AIFF attributes are converted to Sample attributes. These include sustain and release loops, multisample selection, tuning, compression, and data format. Markers other than the sustain and release loops are not stored in the Sample.

The Sample Item is created with { AF_TAG_AUTO_FREE_DATA, TRUE } to facilitate simple and complete deletion. The Sample Item is given the same name as the AIFF file.

When you finish with the sample, you should call `UnloadSample()` to deallocate the resources.

Arguments

fileName

Name of AIFF or AIFC file to load sample from.

Return Value

The procedure returns an item number of the Sample if successful (a non-negative value) or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V27.

Notes

This function is equivalent to:

```
Item LoadSample (const char *fileName)
{
    Item result;
    SampleInfo *smpi = NULL;
    TagArg tags[ML_SAMPLEINFO_MAX_TAGS];

    // Get SampleInfo from AIFF file including the sample data
    if ((result = GetAIFFSampleInfo (&smpi, fileName, 0)) < 0) goto
clean;

    // Get Tags from SampleInfo
    if ((result = SampleInfoToTags (smpi, tags,
ML_SAMPLEINFO_MAX_TAGS)) < 0)
        goto clean;
```



```
        // Create Sample, transfer ownership of data memory to
Sample using
        // AF_TAG_AUTO_FREE_DATA
        if ((result = CreateSampleVA (
            TAG_ITEM_NAME,          fileName,
            AF_TAG_AUTO_FREE_DATA,  TRUE,
            TAG_JUMP,               tags)) < 0) goto clean;

        // Clear allocation flag after successful CreateSample()
call
        // because the Sample is now responsible for freeing it.
        smpi->smpl_Flags &= ~ML_SAMPLEINFO_F_SAMPLE_ALLOCATED;

- . clean:
        DeleteSampleInfo (smpl);
        return result;
    }
```

Module Open Requirements

OpenAudioFolio()

Associated Files

- <:audio:parse_aiff.h>, libmusic.a, libspmath.a, System.m2/Modules/audio,
System.m2/Modules/iff

See Also

UnloadSample(), Sample, CreateAttachment(), GetAIFFSampleInfo(),
SampleInfoToTags(), LoadSystemSample()

LoadSystemSample

Loads a system Sample file.

Synopsis

```
Item LoadSystemSample (const char *fileName)
```

Description

This procedure locates and loads a standard system sample file (e.g., `sinewave.aiff`). These are samples that are part of the operating system to provide simple 'beep' type sounds. Since they are located in `System.m2` they are accessed using `FindFileAndIdentify()`.

Use `UnloadSample()` or `DeleteItem()` to delete the sample item when you are finished with it.

Arguments

```
fileName
    File name of standard system sample file (e.g.,
    sinewave.aiff).
```

Return Value

The procedure returns an item number of the Sample if successful (a non-negative value) or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V30`.

Notes

This function is equivalent to:

```
Item LoadSystemSample (const char *fileName)
{
    char pathName [FILESYSTEM_MAX_PATH_LEN];
    Item result;

    // find file
    if ((result = FindSystemSample (pathName, sizeof pathName,
    fileName)) < 0) return result;

    // attempt to load it
    return LoadSample (pathName);
}

static Err FindSystemSample (char *pathBuf, int32 pathBufSize, const
char *fileName)
{
    static const TagArg fileSearchTags[] = {
        { FILESEARCH_TAG_SEARCH_FILESYSTEMS,
        (TagData)DONT_SEARCH_UNBLESSED },
        TAG_END
    };
};
```

```
static const char sampleDir[] = "System.m2/Audio/aiff/";
char *tempPath;
Err errcode;

    // allocate temp path name
    if (!(tempPath = AllocMem (sizeof sampleDir - 1 + strlen
(fileName) + 1, MEMTYPE_TRACKSIZE))) {
        errcode = ML_ERR_NOMEM;
        goto clean;
    }
    strcpy (tempPath, sampleDir);
    strcat (tempPath, fileName);

    // find file
    errcode = FindFileAndIdentify (pathBuf, pathBufSize, tempPath,
fileSearchTags);

clean:
    FreeMem (tempPath, TRACKED_SIZE);
    return errcode;
}
```

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:parse_aiff.h>, libmusic.a, libspmath.a, System.m2/Modules/audio, System.m2/Modules/iff

See Also

LoadSample(), UnloadSample(), Sample, CreateAttachment()

SampleFormatToInsName

Builds the name of the DSP Instrument Template to play a sample of the specified format.

Synopsis

```
Err SampleFormatToInsName (uint32 compressionType, bool ifVariable,  
                           uint32 numChannels, char *nameBuffer,  
                           int32 nameBufferSize)
```

Description

This procedure builds the name of the instrument that will play a sample of the given format. There is no guarantee that there is an instrument with this name but if there is, it's the right one. This procedure builds the name by concatenating the 4 characters from the compressionType with the fixed or variable flag, and the number of channels. The name will be of the form "sampler_CCCC_PN.dsp" where:

CCCC = the 4 characters of the compression type in lower case, e.g., "sqd2"

P = 'f' for fixed pitch or 'v' for variable pitch

N = the number of channels

For example, an instrument that plays mono ID_SQS2 samples at a fixed rate is: `sampler_sqs2_f1.dsp`.

Arguments

compressionType

The 4 character ID from the AIFC file, or the number of bits (either 8 or 16).

ifVariable

A value indicating whether or not the pitch is to vary. If TRUE, the pitch will vary. If FALSE, the pitch won't vary.

numChannels

The number of channels, 1 for mono, 2 for stereo, and so on, up to 8.

nameBuffer

A character buffer to write the name into.

nameBufferSize

The size of the buffer including the space for the NUL termination.

Return Value

This procedure returns a negative error code or zero.

Implementation

Library call implemented in libmusic.a V27.

Caveats

GIGO: This function will happily return an instrument name for a non-existent instrument (e.g., and unsupported compression type or number of channels).

The matrix of sample player instruments is not complete, so even if all of the individual parameters are

legal, there may not be an instrument to play that format (e.g., `sampler_adp4_v1.dsp` exists but `sampler_adp4_f1.dsp` does not).

Associated Files

<:audio:parse_aiff.h>, libmusic.a, System.m2/Modules/audio

See Also

`SampleItemToInsName()`, `GetAIFFSampleInfo()`, `Sample`

SampleInfoToTags

Fills out a tag list to create a Sample Item from SampleInfo.

Synopsis

```
Err SampleInfoToTags (const SampleInfo *smpi, TagArg *tagBuffer,  
                      int32 maxTags)
```

Description

Fills out a tag list which can be passed to `CreateSample()` to create a Sample item from the SampleInfo.

The sample data pointed to by `smpi_Data` is considered the property of the SampleInfo as long as the `ML_SAMPLEINFO_F_SAMPLE_ALLOCATED` flag is set in `smpi_Flags`. For this reason, this function does not place an `AF_TAG_AUTO_FREE_DATA` tag in the resulting tag list. You may pass this tag along with the tags returned by this function to `CreateSample()`, but be sure to clear the `ML_SAMPLEINFO_F_SAMPLE_ALLOCATED` to avoid doubly freeing the sample data. See below for how to transfer sample data memory ownership to the sample item.

Arguments

- `smpi`
SampleInfo to process.
- `tagBuffer`
TagArg array in which to store resulting tag list.
- `maxTags`
Number of TagArgs in tagBuffer (including space for trailing `TAG_END`).

Return Value

On success, the procedure returns a non-negative value. On failure it returns an error code (a negative value).

On success, `tagBuffer` contains a legal tag list to pass to `CreateSample()`. On failure, `tagBuffer` contents is undefined, and not guaranteed to be a legal list.

Implementation

Library call implemented in `libmusic.a V27`.

Notes

Tag buffer overflow is trapped as an error. To avoid this condition, make sure your TagArg buffer has at least `ML_SAMPLEINFO_MAX_TAGS` elements.

Examples

```
// Code fragment which demonstrates how to create a sample from the  
tags  
// returned by SampleInfoToTags() and transfer the sample data  
memory  
// ownership to the sample item. Be sure to avoid creating more than  
one  
// sample from a given SampleInfo in this manner, or else you will  
get a  
// double free when the sample items are deleted.
```

```
{
    Tags sampleTags[ML_SAMPLEINFO_MAX_TAGS];
    Err errcode;
    Item sampleItem;

    .
    .

    // Get Tags from SampleInfo
    if ((errcode = SampleInfoToTags (sampleInfo, sampleTags,
        ML_SAMPLEINFO_MAX_TAGS)) < 0) goto clean;

    // Create Sample, transfer ownership of data memory to
Sample
    if ((errcode = sampleItem = CreateSampleVA (
        AF_TAG_AUTO_FREE_DATA, TRUE,
        TAG_JUMP, tags)) < 0) goto clean;

    // Data now belongs to Sample, so clear
    // ML_SAMPLEINFO_F_SAMPLE_ALLOCATED to prevent
DeleteSampleInfo()
    // from deleting it.
    sampleInfo->smpi_Flags &= ~ML_SAMPLEINFO_F_SAMPLE_ALLOCATED;

    .
    .
}
```

Associated Files

<:audio:parse_aiff.h>, libmusic.a, System.m2/Modules/audio

See Also

GetAIFFSampleInfo(), LoadSample(), Sample

SampleItemToInsName

Builds the name of the DSP Instrument Template to play the Sample.

Synopsis

```
Err SampleItemToInsName (Item sample, bool ifVariable, char
*nameBuffer,
                        int32 nameBufferSize)
```

Description

This procedure builds the name of the appropriate instrument that will play the given sample item. If there is no appropriate instrument to play the sample, it returns an error code.

This function queries the Sample for its sample rate, number of channels (mono or stereo), sample width (8 or 16), and compression type, and then calls `SampleFormatToInsName()` to get the correct instrument name. It also lets you specify whether you need to vary the pitch or play only at the original pitch.

Arguments

`sample`

The item number of the sample to be played.

`ifVariable`

A value indicating whether or not the pitch is to vary. If TRUE, the pitch will vary. If FALSE, the pitch won't vary.

Setting this to FALSE does not guarantee that you will get a fixed-rate instrument. This function picks an instrument capable of playing the sample at the sample rate stored in the Sample Item. Since all of the fixed-rate sample players play at a sample rate of 44100 Hz, a variable-rate instrument will be chosen to play all samples with an original sample rate other than 44100. A fixed-rate instrument is chosen only if `ifVariable` is FALSE and the sample was recorded at 44100 Hz.

`nameBuffer`

A character buffer to write the name into.

`nameBufferSize`

The size of the buffer including the NUL byte.

Return Value

This procedure returns a negative error code or zero.

Implementation

Library call implemented in `libmusic.a V27`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:parse_aiff.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`SampleFormatToInsName()`, `LoadSample()`, `Sample`

UnloadSample

Unloads a sample loaded by
LoadSample().

Synopsis

```
Err UnloadSample (Item sample)
```

Description

This procedure deletes a sample Item and sample memory for a sample loaded by LoadSample().

Arguments

sample
Item number of sample to delete.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:parse_aiff.h>` V29.

Module Open Requirements

OpenAudioFolio()

Associated Files

`<:audio:parse_aiff.h>`, `System.m2/Modules/audio`

See Also

LoadSample()

ChangeScoreControl

Changes a MIDI control value for a channel.

Synopsis

```
Err ChangeScoreControl( ScoreContext *ScoreCon, int32
                        Channel, int32 Index, int32 Value )
```

Description

This procedure changes a specified MIDI control value for notes played on the specified channel. It's the equivalent of a MIDI control message, currently limited to control values of 7 (channel volume control) and 10 (channel pan control).

The procedure determines the control to change through the value passed in the index argument. It then assigns the value passed in the value argument to the control. For channel volume, the value can range from 0 to 127, with 0 as silence and 127 as maximum volume. For channel panning, the value can range from 0 to 127, with 0 playing all the notes in the left of a stereo output and 127 playing all the notes in the right of a stereo output.

- Arguments

ScoreCon
Pointer to a ScoreContext data structure controlling playback.

Channel
The number of the MIDI channel for which to change the control.

Index
The number of the control to be changed (currently only 7 and 10 are recognized).

Value
A value from 0 to 127 used as the new control setting.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

ChangeScoreProgram(), ChangeScorePitchBend(),
InterpretMIDIMessage(), StartScoreNote(), StopScoreNote()

ChangeScorePitchBend

Changes a channel's pitch bend value.

Synopsis

```
Err ChangeScorePitchBend( ScoreContext *scon,  
                          int32 Channel, int32 Bend )
```

Description

This procedure changes the current MIDI pitch bend value used to alter the pitch values of all instruments in the channel. The procedure is the equivalent of a MIDI Pitch Bend message.

The Bend argument ranges from 0x0 to 0x3FFF. A setting of 0x2000 means that instrument pitch values aren't bent up or down. A setting of 0x0 means that instrument pitches are bent as far down as possible. A setting of 0x3FFF means that instrument pitches are bent as far up as possible.

The pitch bend value supplied by ChangeScorePitchBend() works within a pitch bend range, which is contained in the score context. If the range is 8 semitones, for example, the score voices can be bent up a maximum of 8 semitones or down a minimum of 8 semitones. You can set the pitch bend range using SetScoreBendRange() and read the current pitch bend range using GetScoreBendRange().

Note that ChangeScorePitchBend() only affects notes whose pitches are specified by MIDI note values (0..127). It doesn't affect the pitch of notes specified by frequency.

Arguments

ScoreCon
Pointer to a ScoreContext data structure controlling playback.

Channel
The number of the MIDI channel for which to change the pitch bend value.

Bend
The new pitch bend setting (0x0..0x3FFF).

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V21.

Caveats

Pitch can't be bent beyond the range of the instrument.

Module Open Requirements

OpenAudioFolio()

Associated Files

`<:audio:score.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ChangeScoreControl()`, `ChangeScoreProgram()`, `ConvertPitchBend()`,
`GetScoreBendRange()`, `InterpretMIDIMessage()`, `StartScoreNote()`,
`SetScoreBendRange()`

ChangeScoreProgram

Changes the MIDI program for a channel.

Synopsis

```
Err ChangeScoreProgram( ScoreContext *ScoreCon,
                        int32 Channel, int32 ProgramNum )
```

Description

This procedure changes the current MIDI program used to play notes in a MIDI channel. It's the equivalent of a MIDI program change message.

Arguments

ScoreCon
Pointer to a ScoreContext data structure controlling playback.

Channel
The number of the MIDI channel for which to change the program.

ProgramNum
The number of the new MIDI program to use for the channel.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

ChangeScoreControl(), ChangeScorePitchBend(),
InterpretMIDIMessage(), StartScoreNote(), StopScoreNote()

ConvertPitchBend

Converts a MIDI pitch bend value into frequency multiplier.

Synopsis

```
Err ConvertPitchBend( int32 Bend, int32 SemitoneRange,  
                    float32 *BendFractionPtr )
```

Description

This procedure accepts a MIDI pitch bend value from 0x0 to 0x3FFF. It also accepts a pitch bend range in semitones (half steps). The range value measures the distance from normal pitch to the farthest bent pitch. For example, a pitch bend range of 12 semitones means that an instrument can be bent up in pitch by 12 semitones, and down in pitch by 12 semitones for a total range of 24 semitones (two octaves).

The pitch bend value operates within the pitch bend range: 0x0 means bend pitch all the way to the bottom of the range; 0x2000 means don't bend pitch; and 0x3FFF means bend pitch all the way to the top of the range.

ConvertPitchBend() uses the pitch bend range and the pitch bend value to calculate an internal pitch bend value used to multiply the output of an instrument, bending the instrument's pitch up or down. The internal pitch bend value is written into the variable BendFractionPtr. This bend value is the same as the bend value used for the audio folio call BendInstrumentPitch().

Arguments

Bend

A MIDI pitch bend value from 0x0 to 0x3FFF.

SemitoneRange

A pitch bend range value from 1 to 12, measured in semitones away from normal pitch.

BendFractionPtr

A pointer to a float32 variable in which to store the returned frequency multiplier.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

ChangeScorePitchBend(), GetScoreBendRange(), SetScoreBendRange(),
BendInstrumentPitch()

CreateScoreContext

Allocates a score context.

Synopsis

```
ScoreContext *CreateScoreContext( int32 MaxNumPrograms )
```

Description

This procedure allocates and initializes a score context with room enough for MaxNumPrograms worth of MIDI programs. It also creates a default PIMap for the score context. The score context is necessary to import a MIDI score from a disc file and then play back that file.

Arguments

MaxNumPrograms

The maximum number of MIDI programs (1..128) for which memory will be allocated within the score context.

Return Value

Returns a pointer to a new ScoreContext on success; NULL on failure.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

InitScoreDynamics(), CreateScoreMixer(), ChangeScoreControl()
DeleteScoreMixer(), ChangeScoreProgram(), DeleteScoreContext()

CreateScoreMixer

Creates and initializes a mixer instrument for MIDI score playback.

Synopsis

```
Err CreateScoreMixer( ScoreContext *scon, MixerSpec MixerSpec,  
                     int32 MaxNumVoices, float32 Amplitude )
```

Description

This procedure creates a mixer instrument from the specification and writes the item number of that mixer into the score context so that notes played with that score context are fed through the mixer. The procedure uses the maximum number of voices specified to create an appropriate number of note trackers to handle MIDI note playback. The procedure also uses the amplitude value as the maximum possible amplitude allowed for each voice in the score.

Note that a task should first allocate system amplitude to itself. It can then divide that amplitude up by the number of voices to arrive at a completely safe amplitude value (one that won't drive the DSP to distortion) for this call. Because it's unlikely that all voices will play simultaneously at full amplitude, a task can typically raise amplitude levels above the level considered to be completely safe.

Arguments

scon
Pointer to a ScoreContext data structure.

MixerSpec
Specification for a multi-channel mixer template (as returned by MakeMixerSpec()) to use for this score.

MaxNumVoices
Value indicating the maximum number of voices used for the score. MaxNumVoices cannot exceed the number of inputs on the mixer. If MaxNumVoices is lower than the number of inputs on the mixer, then the higher numbered voices are available for other uses.

Amplitude
Value indicating the maximum volume for each voice in the score. Ranges from 0.0 to 1.0

Return Value

This procedure returns the mixer instrument item if successful, or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

`<:audio:score.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ChangeScoreProgram()`, `InitScoreDynamics()`, `DeleteScoreMixer()`,
`CreateScoreContext()`, `MakeMixerSpec()`

DeleteScoreContext

Deletes a score context.

Synopsis

```
Err DeleteScoreContext( ScoreContext *scon )
```

Description

This procedure deletes the ScoreContext data structure along with all of its attendant data structures, including the NoteTracker data structures used for dynamic voice allocation.

Arguments

scon

Pointer to the ScoreContext data structure to be deleted.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

CreateScoreMixer(), CreateScoreContext()

DeleteScoreMixer

Disposes of mixer created by
CreateScoreMixer().

Synopsis

```
Err DeleteScoreMixer( ScoreContext *scon )
```

Description

This function unloads the mixer loaded by CreateScoreMixer(). This function is automatically called by DeleteScoreContext().

Calling this function multiple times has no harmful effect.

Arguments

scon
 Pointer to a ScoreContext data structure. The mixer need not have been successfully initialized.

Return Value

This procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

CreateScoreMixer(), DeleteScoreContext()

DisableScoreMessages

Enable or disable printed messages during score playback.

Synopsis

```
int32 DisableScoreMessages( int32 Flag )
```

Description

This function turns on or off the informational and error message printing during score playback. Messages default to being enabled.

Arguments

Flag
Non-zero to disable messages, 0 to enable messages.
Messages default to being enabled.

Return Value

Previous enable/disable flag.

Implementation

Library call implemented in libmusic.a V20.

Caveats

This turns off `_most_` score player messages. A few things like MIDI file parsing messages are not controlled by this.

V27 prints fewer messages than V24 and earlier versions.

Associated Files

<:audio:score.h>, libmusic.a

FreeChannelInstruments

Frees all of a MIDI channel's instruments.

Synopsis

```
Err FreeChannelInstruments( ScoreContext *scon,  
                           int32 Channel )
```

Description

This procedure frees all instruments currently allocated to a MIDI channel of a score context, an action that abruptly stops any notes the instruments may be playing.

This call is useful because allocated instruments may accumulate in a channel as dynamic voice allocation creates instruments to handle simultaneous note playback within the channel. As notes stop, the instruments used to play the notes are abandoned, but not freed, remaining in existence so they can be used to immediately play incoming notes. These abandoned instruments use up system resources.

If a task knows that it will not play notes in a MIDI channel for a period of time, it can use FreeChannelInstruments() to free all allocated instruments dedicated to a channel. This frees all of the system resources used to support those instruments, making them available for other audio jobs. When notes are played in a channel after all of its voices are freed, dynamic voice allocation immediately allocates new instruments to play those notes.

Arguments

scon
 Pointer to a ScoreContext data structure.

Channel
 The number of the MIDI channel for which to free instruments.

Return Value

This macro returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

ReleaseScoreNote(), StartScoreNote()

GetScoreBendRange

Gets the current pitch bend range value for a score context.

Synopsis

```
int32 GetScoreBendRange( ScoreContext *scon )
```

Description

This procedure retrieves the pitch bend range value currently set for the score. This value is an integer, measured in semitones (half steps), that sets the range above or below normal that instruments can be bent. A pitch range of two, for example, means that instruments can bend up two semitones (a whole step up) and down two semitones (a whole step down).

Arguments

scon
Pointer to a ScoreContext data structure.

Return Value

This procedure returns the current pitch bend range value of the specified score if successful, or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

ChangeScorePitchBend(), ConvertPitchBend(), SetScoreBendRange()

InitScoreDynamics

Sets up dynamic voice allocation.

Synopsis

```
Err InitScoreDynamics( ScoreContext *scon,  
                      int32 MaxScoreVoices )
```

Description

This procedure creates an appropriate number of note trackers to handle the maximum number of voices specified. It's called internally by CreateScoreMixer(). You should use this call instead of CreateScoreMixer() if your task is setting up MIDI score playback using non-DSP instruments. InitScoreDynamics() sets up voice allocation without creating a mixer instrument.

Arguments

scon
Pointer to a ScoreContext data structure.

MaxNumVoices
A value indicating the maximum number of voices for the score.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

CreateScoreMixer(), DeleteScoreMixer()

InterpretMIDIEvent

Interprets MIDI events within a MIDI object.

Synopsis

```
Err InterpretMIDIEvent( Sequence *SeqPtr, MIDIEvent *MEvCur,  
                      ScoreContext *scon )
```

Description

This procedure is the interpreter procedure for MIDI sequence objects created using `MFLoadSequence()` and `MFLoadCollection()`. It's an internal procedure that is called by the juggler and, in turn, calls `InterpretMIDIMessage()`.

Arguments

`SeqPtr`
Pointer to a sequence data structure.

`MEvCur`
Pointer to the current MIDI event within the sequence.

`scon`
Pointer to a `ScoreContext` data structure used for playback.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V20`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:patchfile.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`InterpretMIDIMessage()`

InterpretMIDIMessage

Executes a MIDI message.

Synopsis

```
Err InterpretMIDIMessage( ScoreContext *ScoreCon,  
                          uint8 *MIDIMsg, int32 IfMute )
```

Description

This procedure executes a MIDI message by calling another music library procedure appropriate to that message, and passing the message's data to that procedure. The called procedure uses audio folio procedures for playback based on the settings of a ScoreContext data structure.

InterpretMIDIMessage() is called by InterpretMIDIEvent(), which extracts MIDI messages from juggler sequences containing MIDI events. Although this procedure is used most often as an internal call of the music library, tasks may call it directly to execute a supplied MIDI message. The message should be stored in the first byte (first character) of the MIDIMsg string, followed by data bytes (if present) in subsequent bytes of the string.

Whenever the IfMute argument is set to TRUE, this procedure does not process Note On messages with velocity values greater than zero. In other words, it doesn't start new notes, but it does release existing notes and process all other recognized MIDI messages. When IfMute is set to FALSE, this procedure processes all recognizable messages, including Note On messages.

Note that InterpretMIDIEvent() reads a juggler sequence's mute flag and, if true, passes that true setting on to InterpretMIDIMessage() so that the sequence stops playing notes. It likewise passes a false setting on to InterpretMIDIMessage() so that the sequence can play notes.

Arguments

ScoreCon
Pointer to a ScoreContext data structure.

MIDIMsg
Pointer to a character string containing the MIDI message.

IfMute
A flag that turns muting on or off. TRUE is on, FALSE is off.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Caveats

This procedure only handles Note On, Note Off, Change Program, Change Control (values 7 and 10), and Pitchbend MIDI messages. It does not support running status.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

InterpretMIDIEvent()

LoadPIMap

Loads a Program-Instrument Map (PIMap) from a text file.

Synopsis

```
Err LoadPIMap (ScoreContext *scon, const char *fileName)
```

Description

This procedure reads the designated Program-Instrument Map file (PIMap file) and writes appropriate values to the specified score context's PIMap. It also assigns appropriate sampled-sound instruments to samples listed in the PIMap file and imports all instrument templates listed in the PIMap file.

For information about the format of a PIMap file, read the "Playing MIDI Scores" chapter of the Portfolio Programmer's Guide.

Note that if you want to set a score context's PIMap entries directly, you can use SetPIMapEntry().

Arguments

scon
Pointer to a ScoreContext data structure.

fileName
Pointer to the character string containing the name of the PIMap file.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:patchfile.h>, libmusic.a, libspmath.a, System.m2/Modules/audio, System.m2/Modules/audiopatchfile, System.m2/Modules/iff

See Also

SetPIMapEntry(), UnloadPIMap(), PIMap

LoadScoreTemplate

Loads instrument or patch Template depending on file name extension.

Synopsis

```
Item LoadScoreTemplate (const char *fileName)
```

Description

This function examines the fileName extension to determine which of the following ways to load and return an Instrument Template:

```
.dsp
    LoadInsTemplate()

.patch
    LoadPatchTemplate()
```

The score player uses this, which explains its grouping and name, but it can be used independently from the score player.

You may use UnloadScoreTemplate() to dispose of the Template when done with it.

Arguments

fileName
The name of the DSP Instrument or Patch Template to load. In the case of standard DSP Instrument Templates (e.g., sawtooth.dsp), there should be no directory component to the file name. Full path names are supported for patch file names.

Return Value

Returns a Template Item number (non-negative value) if successful, or an error code (negative value) if an error occurs.

Implementation

Convenience call implemented in libmusic.a V29.

Notes

This function is equivalent to:

```
Item LoadScoreTemplate (const char *fileName)
{
    const char * const suffix = strrchr (fileName, '.');

    return (suffix && !strcasecmp(suffix, ".patch"))
        ? internalLoadPatchTemplate (fileName)
        : LoadInsTemplate (fileName, NULL);
}

static Item internalLoadPatchTemplate (const char *fileName)
{
    Item result;
```

```
    if ((result = OpenAudioPatchFileFolio()) >= 0) {  
        result = LoadPatchTemplate (fileName);  
        CloseAudioPatchFileFolio();  
    }  
  
    return result;  
}
```

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio,
System.m2/Modules/audiopatchfile

See Also

LoadInsTemplate(), LoadPatchTemplate(), UnloadScoreTemplate(),
Template

MFDefineCollection

Creates a juggler collection from a MIDI file image in RAM.

Synopsis

```
Err MFDefineCollection( MIDIFileParser *mfpptr, uint8 *Image,  
                        int32 NumBytes, Collection *ColPtr)
```

Description

This procedure creates a juggler collection from a MIDI file image imported into RAM as part of a data streaming process. The MIDI file in the image may be a format 0 or a format 1 MIDI file. To use this procedure, you must first create a juggler collection using `CreateObject()`

. The procedure then creates an appropriate number of juggler sequences to contain the sequences found in the MIDI file image.

The procedure translates the MIDI messages in each file sequence into MIDI events in an appropriate juggler sequences within the collection. To work, the procedure must have a `MIDIFileParser` data structure to keep track of the MIDI sequence's original settings.

Note that this procedure treats format 0 MIDI files as one-track format 1 MIDI files.

Arguments

`mfpptr`
Pointer to a `MIDIFileParser` data structure.

`Image`
Pointer to a MIDI file image in RAM (a string of bytes).

`NumBytes`
The size of the MIDI file image in bytes.

`ColPtr`
Pointer to a juggler collection in which to store the converted MIDI sequences.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a` V20.

Associated Files

<:audio:score.h>, `libmusic.a`

See Also

`CreateObject()`, `MFLoadCollection()`, `MFLoadSequence()`,
`MFUnloadCollection()`

MFLoadCollection

Loads a set of sequences from a MIDI file.

Synopsis

```
Err MFLoadCollection( MIDIFileParser *mfpptr, char *filename,  
                      Collection *ColPtr )
```

Description

This procedure loads a set of sequences from a format 1 MIDI file and turns them into a juggler collection for playback using the juggler. To use this procedure, you must first create a juggler collection using `CreateObject()`. The procedure then creates an appropriate number of juggler sequences to contain the sequences found in the MIDI file.

The procedure translates the MIDI messages in each file sequence into MIDI events in appropriate juggler sequences within the collection. To work, the procedure must have a `MIDIFileParser` data structure to keep track of the MIDI sequence's original settings.

Note that this procedure also accept format 0 MIDI files, treating them as a one-track format 1 MIDI file.

Arguments

`mfpptr`
Pointer to a `MIDIFileParser` data structure.

`filename`
Pointer to a character string containing the name of the format 1 MIDI file from which to load the sequences.

`ColPtr`
Pointer to a juggler collection in which to store the converted MIDI sequences.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a` V20.

Associated Files

<:audio:score.h>, `libmusic.a`

See Also

`CreateObject()`, `MFLoadSequence()`, `MFUnloadCollection()`

MFLoadSequence

Loads a sequence from a MIDI file.

Synopsis

```
Err MFLoadSequence( MIDIFileParser *mfpptr, char *filename,  
                    Sequence *SeqPtr )
```

Description

This procedure loads a single sequence from a format 0 MIDI file and turns it into a juggler sequence for playback using the juggler. To use this procedure, you must first create the juggler sequence using `CreateObject()`. The procedure translates the MIDI messages in the sequence into MIDI events in the juggler sequence. To work, the procedure must have a `MIDIFileParser` data structure to keep track of the MIDI sequence's original settings.

Note that this procedure will accept a format 1 MIDI file, in which case it converts only the first track (sequence) of the file. That track typically contains no notes.

Arguments

`mfpptr`
Pointer to a `MIDIFileParser` data structure.

`filename`
Pointer to a character string containing the name of the format 0 MIDI file from which to load the sequence.

`SeqPtr`
Pointer to a juggler sequence in which to store the converted MIDI sequence.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V20`.

Associated Files

`<:audio:score.h>`, `libmusic.a`

See Also

`CreateObject()`, `MFLoadCollection()`, `MFUnloadCollection()`

MFUnloadCollection

Unloads a MIDI collection.

Synopsis

```
Err MFUnloadCollection( Collection *ColPtr )
```

Description

This procedure unloads a juggler collection created by importing a MIDI format 1 file. Unloading destroys the collection and any sequences defined as part of it.

Arguments

ColPtr
 Pointer to a juggler collection data structure.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

MFLoadCollection(), MFLoadSequence()

NoteOffIns

Turns off a note played by an instrument.

Synopsis

```
Err NoteOffIns( Item Instrument, int32 Note, int32 Velocity )
```

Description

This procedure releases a note being played by an instrument and specifies the pitch and velocity for the instrument to release the note. The procedure does not use voice allocation for the note. It's called by ReleaseScoreNote() after ReleaseScoreNote() has worked out voice allocation.

Arguments

Instrument

The item number of the instrument.

Note

The MIDI pitch value of the note.

Velocity

The MIDI release velocity value of the note.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Caveats

At present, release velocity is ignored.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

NoteOnIns(), ReleaseScoreNote(), StartScoreNote()

NoteOnIns

Turns on a note for an instrument.

Synopsis

```
Err NoteOnIns( Item Instrument, int32 Note, int32 Velocity )
```

Description

This procedure turns on a note for an instrument and specifies the pitch and velocity for the instrument to play the note. The procedure does not use voice allocation for the note. It's called by StartScoreNote() after StartScoreNote() has worked out voice allocation.

Arguments

Instrument

The item number of the instrument.

Note

The MIDI pitch value of the note.

Velocity

The MIDI attack velocity value of the note. If attack velocity is 0, the specified note is released (identical behavior to the MIDI Note On event).

Return Value

This procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

NoteOffIns(), ReleaseScoreNote(), StartScoreNote()

PIMap

Program-Instrument Map file format.

Description

A PIMap file consists of any number of lines formatted as below. Comments must have a semicolon in the first character of the line. Blank lines are ignored.

Format

<program number> <name> [switches]

Arguments

These arguments are required for each line of a PIMap.

<program number>

MIDI program number to associate <name> with. The valid range is 1..128. The highest number allowed is determined from MaxNumPrograms argument of CreateScoreContext() for the ScoreContext to be associated with the PIMap.

<name>

Name of a sample, DSP instrument template, or patch template to associate with this program number. There can be multiple samples associated with a program number, but only one instrument or patch.

If the first occurrence of a program number has a patch or instrument name, that patch or instrument is associated with the program number. Subsequent samples can be associated with this program number, and are automatically attached to the instrument or patch template.

If the first occurrence of a program number has a sample name (file name extension is .aiff, .aifc, or .aif), then a suitable template is automatically chosen to play the sample. Subsequent samples can be attached to this template just as with patches and instrument templates.

Switches

Any number of these optional switches may appear after the required arguments.

-b <MIDI note number>

Sets base note of sample. Range is 0..127. Defaults to value from sample file.

-d <cents>

Detune value for sample in cents. Range is -100..100. Defaults to value from sample file.

-f

Causes automatic sample player template selection to use a fixed-rate template. If not specified, a variable-rate sample

player is selected.

-h <MIDI note number>

Sets upper note limit for a sample belonging to a multi-sample program number. Range is 0..127. Defaults to value from sample file.

-hook <name>

Name of FIFO hook to attach sample to for instruments with multiple FIFOs.

-l <MIDI note number>

Sets lower note limit for a sample belonging to a multi-sample program number. Range is 0..127. Defaults to value from sample file.

-m <num voices>

Specifies the maximum number of voices to assign to this program number. Range is 1..127. Defaults to 1.

-p <priority>

Sets instrument priority for this program number. Range is 0..200. Defaults to 100.

-r <rate divisor>

Specifies the instrument execution rate division for this program number. Valid settings are 1, 2, and 8. Defaults to 1. See CreateInstrument() for more information on this.

See Also

LoadPIMap(), CreateInstrument(), LoadSample(), "Playing MIDI Scores" chapter of the Portfolio Programmer's Guide.

PurgeScoreInstrument

Purges an unused instrument from a ScoreContext.

Synopsis

```
Err PurgeScoreInstrument( ScoreContext *scon, uint8 Priority, int32
MaxLevel )
```

Description

This procedure purges an unused instrument from a ScoreContext. This call, coupled with the scon_PurgeHook allows an application to share DSP resources with the ScoreContext.

Arguments

scon
A pointer to a ScoreContext data structure controlling playback.

Priority
The maximum instrument priority to purge for instruments that are still playing (in the range of 0 to 200). Instruments of higher priority than this may be purged if they have stopped playing.

MaxLevel
The maximum activity to purge (i.e. AF_ABANDONED, AF_STOPPED, AF_RELEASED, AF_STARTED).

Return Value

This procedure returns a positive value if an instrument was actually purged, zero if no instrument matching the specifications could be purged, or a negative error code on failure.

Examples

```
// This code fragment can be used to free the DSP resources used by
all
// instruments that have finished playing:

{
    int32 result;

    // loop until function returns no voice purged or error
    while ( (result = PurgeScoreInstrument (scon,
SCORE_MAX_PRIORITY,
                                           AF_STOPPED)) > 0 ) ;

    // catch error
    if (result < 0) ...
}
```

Implementation

Library call implemented in libmusic.a V20.

Caveats

This function deletes items that are created by `StartScoreNote()`. Frequent use of this function and `StartScoreNote()` can consume the item table. If you simply want to stop a score note in it's tracks for later use with the same channel, use `StopScoreNote()` instead. It merely stops instruments and doesn't delete them.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:score.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`StopScoreNote()`

ReleaseScoreNote

Releases a MIDI note, uses voice allocation.

Synopsis

```
Err ReleaseScoreNote( ScoreContext *scon, int32 Channel,  
                      int32 Note, int32 Velocity )
```

Description

This procedure releases a note for the score context and uses dynamic voice allocation to do so. It is equivalent to a MIDI NoteOff message.

Arguments

scon

A pointer to a ScoreContext data structure controlling playback.

Channel

The number of the MIDI channel in which to play the note.

Note

The MIDI pitch value of the note (0..127).

Velocity

The MIDI release velocity value of the note (0..127).

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Caveats

At present, release velocity is ignored.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

FreeChannelInstruments(), NoteOnIns(), NoteOffIns(),
StartScoreNote(), StopScoreNote()

SetPIMapEntry

Specifies the instrument to use when a MIDI program change occurs.

Synopsis

```
Err SetPIMapEntry( ScoreContext *scon, int32 ProgramNum,  
                  Item InsTemplate, int32 MaxVoices, int32 Priority  
                )
```

Description

This procedure specifies the instrument type to use when a MIDI program change occurs.

This procedure directly sets a PIMap entry, assigning an instrument template, a maximum voice value, and an instrument priority to a MIDI program number. When a MIDI program change occurs, a new instrument specified by the instrument template is used for notes played after the program change.

The variable "ProgramNum" ranges from 0 to 127 (unlike some synthesizers, which range program numbers from 1 to 128). It must not exceed the number of programs allocated in CreateScoreContext().

Note that it's easier to create a full PIMap using LoadPIMap().

Arguments

scon

A pointer to the ScoreContext data structure whose PIMap is to be changed.

ProgramNum

Value indicating a program number, ranging from 0..127; not to exceed the maximum number allocated in CreateScoreContext().

InsTemplate

The item number of an instrument template.

MaxVoices

The maximum number of voices that can play simultaneously for this program.

Priority

An instrument priority value from 0 to 200.

Return Value

This procedure returns 0 if all went well; or an error code (less than 0) indicating that an error occurred.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

LoadPIMap(), UnloadPIMap()

SetScoreBendRange

Sets the current pitch bend range value for a score context.

Synopsis

```
Err SetScoreBendRange( ScoreContext *scon, int32 BendRange )
```

Description

The procedure sets a score's current pitch bend range value. This value is an integer, measured in semitones (half steps), that sets the range above or below normal that instruments can be bent. A pitch range of two, for example, means that instruments can bend up two semitones (a whole step up) and down two semitones (a whole step down).

Note that DSP sampled-sound instruments won't bend any more than 12 semitones, so don't set BendRange to a value greater than 12.

Arguments

`scon`
Pointer to a ScoreContext data structure.

`BendRange`
A pitch bend range value from 1 to 12.

Return Value

This procedure returns 0 if all went well; otherwise, an error code (less than 0) indicating that an error occurred.

Implementation

Library call implemented in libmusic.a V20.

Associated Files

<:audio:score.h>, libmusic.a

See Also

ChangeScorePitchBend(), ConvertPitchBend(), GetScoreBendRange()

StartScoreNote

Starts a MIDI note, uses voice allocation.

Synopsis

```
Err StartScoreNote( ScoreContext *scon, int32 Channel,
                   int32 Note, int32 Velocity )
```

Description

This procedure turns on a note for the score context and uses dynamic voice allocation to do so. It is equivalent to a MIDI Note On message.

Arguments

scon

A pointer to a ScoreContext data structure controlling playback.

Channel

The number of the MIDI channel in which to play the note.

Note

The MIDI pitch value of the note (0..127).

Velocity

The MIDI attack velocity value of the note (0..127). If attack velocity is 0, the specified note is released (identical behavior to the MIDI Note On event).

Return Value

This procedure returns a non-negative value if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

FreeChannelInstruments(), NoteOnIns(), NoteOffIns(),
ReleaseScoreNote(), StopScoreNote()

StopScoreNote

Stop a MIDI note immediately with no release phase.

Synopsis

```
Err StopScoreNote( ScoreContext *scon, int32 Channel, int32 Note )
```

Description

This procedure immediately stops a note for the score context. It differs from ReleaseScoreNote() in that it doesn't allow the note to go through any release phase that it might have.

Arguments

scon

A pointer to a ScoreContext data structure controlling playback.

Channel

The number of the MIDI channel in which to play the note.

Note

The MIDI pitch value of the note (0..127).

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V24.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:score.h>, libmusic.a, System.m2/Modules/audio

See Also

FreeChannelInstruments(), NoteOnIns(), NoteOffIns(),
StartScoreNote(), ReleaseScoreNote(), PurgeScoreInstrument()

UnloadPIMap

Unloads instrument templates loaded previously with PIMap file.

Synopsis

```
Err UnloadPIMap( ScoreContext *scon )
```

Description

This procedure is the inverse of LoadPIMap; it unloads any instrument templates specified in the score context's PIMap. Any instruments created using those templates are freed when the templates are unloaded.

Arguments

scon
Pointer to a ScoreContext data structure.

Return Value

This procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V20.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:patchfile.h>, libmusic.a, System.m2/Modules/audio

See Also

LoadPIMap(), SetPIMapEntry()

UnloadScoreTemplate

Unloads a Template loaded by
LoadScoreTemplate().

Synopsis

```
Err UnloadScoreTemplate (Item instTemplate)
```

Description

This macro deletes an instrument Template loaded by
LoadScoreTemplate().

Arguments

```
instTemplate  
    Instrument Template Item to unload.
```

Return Value

Returns a non-negative value if successful, or an error code
(negative value) if an error occurs.

Implementation

Macro implemented in `<:audio:score.h>` V29.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

`<:audio:score.h>`, `System.m2/Modules/audio`

See Also

```
LoadScoreTemplate()
```

Create3DSound

Allocate and initialize the 3D Sound data structures.

Synopsis

```
Err Create3DSound( Sound3D** s3dHandle, const TagArg* tagList )
```

```
Err Create3DSoundVA( Sound3D** s3dHandle, uint32 tag1, ... )
```

Description

Create3DSound() allocates DSP resources required to implement the 3D cues selected with the S3D_TAG_FLAGS tag, currently the only valid tag in the tagList function argument. A pointer to the resulting structure is returned in the variable pointed to by the argument s3dHandle. The pointer is subsequently passed to other functions in the 3DSound API.

Use Delete3DSound() to free all the resources associated with the sound.

Arguments

s3dHandle

pointer to location used to store pointer to 3DSound structure.

tagList

a pointer to an array of tags.

Tags

S3D_TAG_FLAGS (uint32)

Specifies which of the 3D cues that 3DSound supports are to be used for this particular sound. The flags should be logically OR-ed together to form the tag argument, from the list below.

If this tag is not specified, a default set of flags will be used, consisting of S3D_F_PAN_DELAY | S3D_F_PAN_FILTER | S3D_F_DOPPLER | S3D_F_DISTANCE_AMPLITUDE_SQUARE | S3D_F_OUTPUT_HEADPHONES.

S3D_TAG_BACK_AMPLITUDE (float32)

Sounds may be specified as being omni-directional, where the sound radiates equally in all directions, or directional, where the sound "points" in a particular direction. In the latter case, the S3D_TAG_BACK_AMPLITUDE tag should be specified.

The argument value represents a "backward amplitude" factor. When this is 1.0, the amplitude of the sound radiated behind the source is the same as that in front, so you get an omnidirectional response. When it's 0.0, the amplitude directly behind the source falls to 0, according to a modified cardioid equation given below.

Using an intermediate value results in some sound being heard from any angle around the source.

If this tag is not specified, the default value is 1.0 (an omnidirectional source).

Unidirectional Cardioid Equation

$$\text{amp} = \frac{\text{frontamp}(1.0 - ((1.0 - \text{backamp}) * \cos(\text{theta}))}{(2.0 - \text{backamp})}$$

where theta is the difference angle between the direction of radiation in 3-space and a line from the sound to the observer.

Flags

S3D_F_PAN_DELAY

Uses a delay line to simulate interaural time differences (ITD). For most sounds, this provides a better azimuth cue than simply adjusting the left and right channel gain, at the expense of three FIFOs. Can be used with S3D_F_PAN_AMPLITUDE to provide interaural intensity differences, but this is usually better done by pairing with S3D_F_PAN_FILTER.

S3D_F_PAN_AMPLITUDE

Uses gain control to simulate interaural intensity differences (IID). For steady-state sounds with spectral content mostly above 1.5K, this provides a reasonable azimuth cue with low DSP overhead. When used with S3D_F_PAN_FILTER tends to over-emphasize interaural gain (left/right) differences, but by itself provides no front/back or up/down cues.

S3D_F_PAN_FILTER

Uses a custom filter instrument to simulate head- and upper-body-related effects. In combination with either the PAN_DELAY or PAN_AMPLITUDE flag, this provides much better azimuth cues, especially for differentiating between front and back.

S3D_F_DOPPLER

This flag indicates that the 3DSound should calculate frequency shifts due to the Doppler effect when the sound is moving. Frequency information is passed back to the application through the Get3DSoundParms function, where it may be used to control frequency-dependent aspects of the source instrument (the sample rate, lfo rates and so on). Primarily useful for fast-moving sounds. The doppler calculation doesn't take any DSP resources, but utilizes the host CPU.

S3D_F_DISTANCE_AMPLITUDE_SQUARE

This flag indicates that amplitude should be calculated according to the inverse square law, whereby intensity drops off

proportional to the square of the distance between the sound and the observer. While physically correct, studies show that for some sounds, particularly familiar ones, an inverse cubed law may sound more realistic. The amplitude information is passed back to the application through the Get3DSoundParms function as a "distance factor" from 1.0 to 0.0, where 1.0 is the closest and 0.0 the farthest away. The application may use the number to directly control the gain of the sound in the final mix, as well as control the amount of reverberance of the sound. The distance factor calculation doesn't take any DSP resources.

S3D_F_DISTANCE_AMPLITUDE_SONE

This flag indicates that amplitude should be calculated according to an approximation of the "sone" scale, where intensity drops off according to an inverse cube relationship to distance between sound and observer. Frequency-dependent amplitude attenuation is not enabled with this flag, but rather is done with a low-pass filter when using the S3D_F_PAN_FILTER flag. The amplitude information is passed back to the application through the Get3DSoundParms function as a "distance factor" from 1.0 to 0.0, where 1.0 is the closest and 0.0 the farthest away. The application may use the number to directly control the gain of the sound in the final mix, as well as control the amount of reverberance of the sound. The distance factor calculation doesn't take any DSP resources.

S3D_F_OUTPUT_HEADPHONES

If this flag is not set, it is assumed that the observer will be hearing sound over loudspeakers positioned to either side of the video display. In this case, many of the 3D sound cues are inaudible, and so will not be used regardless of the settings of other flags. Additionally, cross-channel cancellation will be employed to decorrelate signals from each of the loudspeakers.

S3D_F_SMOOTH_AMPLITUDE

When you select PAN_AMPLITUDE, or you use the BACK_AMPLITUDE tag for a unidirectional sound source, Sound3D updates an gain scalar every time Move3DSound is called. In some situations, this can result in audible quantizing of the sound's loudness. If this happens, you can smooth out the gain changes by setting this flag. The smoothing uses the envelope.dsp instrument, so there's a penalty in dsp resource usage associated with it. If you're running low on resources, try disabling this flag.

The default is disabled.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V30.

Caveats

Elevation cues are not yet supported.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio,
System.m2/Modules/audiopatch

See Also

Delete3DSound(), Start3DSound(), Stop3DSound(), Move3DSound(),
Move3DSoundTo(), Get3DSoundInstrument(), Get3DSoundPos(),
Get3DSoundParms()

Delete3DSound

Release resources and deallocate 3D Sound data structures.

Synopsis

```
void Delete3DSound( Sound3D** a3DSound )
```

Description

This function stops a 3D Sound playing (if it's playing), deletes all the items associated with the sound and frees the DSP resources and memory associated with the sound. It does not delete or otherwise affect the source sound connected to the 3D Sound.

After freeing memory, it clears the pointer in the handle a3DSound to NULL.

Arguments

a3DSound
a pointer to a pointer to a Sound3D structure, previously set using a call to Create3DSound().

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Start3DSound(), Stop3DSound(), Move3DSound(),
Move3DSoundTo(), Get3DSoundInstrument(), Get3DSoundPos(),
Get3DSoundParms()

Get3DSoundInstrument

Returns the DSP instrument to which a source should be connected to use a 3D Sound.

Synopsis

```
Item Get3DSoundInstrument( Sound3D* a3DSound )
```

Description

The 3D Sound calls process the output of a source instrument to provide an illusion of space. The source instrument (or patch) is created, destroyed and otherwise maintained by the application. In order to connect the source instrument to the 3D Sound, the application uses this function to find the 3D Sound instrument that can subsequently be used in a call to ConnectInstrument().

Arguments

a3DSound
a pointer to a Sound3D structure, previously set using a call to Create3DSound().

Return Value

The procedure returns the item number of the 3D Sound instrument if successful, or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Delete3DSound(), Start3DSound(), Stop3DSound(), Move3DSound(), Move3DSoundTo(), Get3DSoundPos(), Get3DSoundParms()

Get3DSoundParms

Return application-dependent cue parameters.

Synopsis

```
Err Get3DSoundParms( Sound3D* a3DSound, Sound3DParms* Snd3DParms,
                    uint32 s3dParmsSize )
```

Description

Some spatialization cues are implemented by the 3D Sound. Others rely on the application to implement. For example, doppler shifting may be implemented by having the application modify frequency components of the source sound instrument, or various modifications to the reverberation characteristics of the sound based on the perceived distance from the observer to the sound may be implemented by a reverb/mix patch in the application.

The application needs information from the 3D Sound in order to implement these cues. This function is used to get that information, by filling in the Sound3DParms structure pointed to. The values are recalculated on calls to Move3DSound(), Move3DSoundTo() and Start3DSound(), so generally this function will be called immediately following one of those.

The Sound3DParms structure contains the following fields:

float32 s3dp_Doppler
a frequency scaling factor. 1.0 implies no change in frequency (a relative velocity of 0). Less than 1.0 implies a lower frequency (the object is moving away). Greater than 1.0 implies a higher frequency (the object is moving toward the observer). Generally, multiplying the "Frequency" or "SampleRate" of the source instrument by this number will give a good approximation of doppler shift.

float32 s3dp_DistanceFactor
ranges from 1.0 when the sound is located directly beside or inside the head, to 0.0 when the sound is maximally distant. Intermediate distances are calculated according to the distance/ amplitude cue selected with Create3DSound(): either proportional to 1/distance (using S3D_F_DISTANCE_AMPLITUDE_SQUARE) or to 1/(distance**3/2) (using S3D_F_DISTANCE_AMPLITUDE_SONE). Using this number to control the gain of the sound in the output mix provides amplitude cues; it can also be used to calculate the wet/dry mix in a reverberant space.

If the s3dParmsSize argument is less than the size of a Sound3DParms structure, only the first s3dParmsSize bytes are copied.

Arguments

a3DSound
a pointer to a Sound3D structure, previously set using a call to Create3DSound().

Snd3DParms

a pointer to a Sound3DParms structure to be filled in by this function.

s3dParmsSize

the size of the Sound3DParms structure to be filled in.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Delete3DSound(), Start3DSound(), Stop3DSound(),
Move3DSound(), Move3DSoundTo(), Get3DSoundInstrument(),
Get3DSoundPos()

Get3DSoundPos

Calculate the instantaneous position of a 3D Sound at the time of the call.

Synopsis

```
Err Get3DSoundPos( Sound3D* a3DSound, PolarPosition4D* Pos4D )
```

Description

This function calculates the current position of a 3D Sound from the values of the cues that the sound is using for spatialization (specified when the sound was created using `Create3DSound()`). The current audio frame count is retrieved, and the sound's polar position calculated.

Obviously, if some cues are not being used, corresponding position information will be meaningless. For example, if neither `S3D_F_PAN_DELAY` or `S3D_F_PAN_AMPLITUDE` were specified when the sound was created, there is no way to derive the sound's azimuth angle.

The radius is derived from a linear interpolation between the last-used start and end positions.

Arguments

`a3DSound`

a pointer to a `Sound3D` structure, previously set using a call to `Create3DSound()`.

`Pos4D`

a pointer to a `PolarPosition4D` structure to be filled in by this function with the calculated current position and time of the sound. See `<:audio:sound3d.h>`.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V30`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:sound3d.h>`, `libmusic.a`, `libspmath.a`, `System.m2/Modules/audio`

See Also

`Create3DSound()`, `Delete3DSound()`, `Start3DSound()`, `Stop3DSound()`,
`Move3DSound()`, `Move3DSoundTo()`, `Get3DSoundInstrument()`,
`Get3DSoundParms()`

Move3DSound

Moves a 3D Sound in a line from start to end over a given time period.

Synopsis

```
Err Move3DSound( Sound3D* a3DSound, PolarPosition4D* Start4D,  
                PolarPosition4D* End4D )
```

Description

This function animates a sound, moving it from the position specified in the argument Start4D to that in End4D, over a time period given by (End4D.pp4d_Time - Start4D.pp4d_Time). The illusion of movement is generated using the cues specified when the 3D Sound was created with Create3DSound(). Applications that use host-level cues (for example, doppler shift using frequency changes, distance-based amplitude and reverberation) can call Get3DSoundParms() immediately after calling this function to retrieve current values for these cues.

Since time is specified in frame counts and some parameters are set directly rather than ramped, this function should be called relatively frequently (at least several times a second) for a moving sound. This is especially important for sounds close to the observer, since the angles subtended by the movement may be quite large.

Arguments

a3DSound

a pointer to a Sound3D structure, previously set using a call to Create3DSound().

Start4D

the position and time vector at the start of the sound's movement. See <:audio:sound3d.h>.

End4D

the position and time vector to be moved to.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Delete3DSound(), Start3DSound(), Stop3DSound(),
Move3DSoundTo(), Get3DSoundInstrument(), Get3DSoundPos(),
Get3DSoundParms()

Move3DSoundTo

Moves a 3D Sound in a line from wherever it currently is to a given endpoint.

Synopsis

```
Err Move3DSoundTo( Sound3D* a3DSound, PolarPosition4D* Target4D )
```

Description

This function animates a sound, moving it from its current position (as returned by a call to `Get3DSoundPos()`) to that in `Target4D` over a time period given by `(Target4D.pp4d_Time - GetAudioFrameCount())`. The illusion of movement is generated using the cues specified when the 3D Sound was created using `Create3DSound()`. Applications that use host-level cues (for example, doppler shift using frequency changes, distance-based amplitude and reverberation) can call `Get3DSoundParms()` immediately after calling this function to retrieve current values for these cues.

This function is equivalent to calling `Get3DSoundPos()`, and using the result as the `Start4D` argument in a call to `Move3DSound()`.

Arguments

`a3DSound`

a pointer to a `Sound3D` structure, previously set using a call to `Create3DSound()`.

`Target4D`

the position and time vector to be moved to.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a V30`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:sound3d.h>`, `libmusic.a`, `libspmath.a`, `System.m2/Modules/audio`

See Also

`Create3DSound()`, `Delete3DSound()`, `Start3DSound()`, `Stop3DSound()`,
`Move3DSound()`, `Get3DSoundInstrument()`, `Get3DSoundPos()`,
`Get3DSoundParms()`

s3dNormalizeAngle

Normalize an angle to be between $-\pi$ and $+\pi$.

Synopsis

```
float32 s3dNormalizeAngle( float32 Angle )
```

Description

This functions adds or subtracts 2π so that the resulting angle is between negative π and positive π .

Arguments

Angle

an angle expressed in radians. Angle must be between -3π and $+3\pi$ or the resulting angle will not be normalized.

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Start3DSound(), Stop3DSound(), Move3DSound(),
Move3DSoundTo(), Get3DSoundInstrument(), Get3DSoundPos(),
Get3DSoundParms()

Start3DSound

Starts a 3D Sound, and positions it in space.

Synopsis

```
Err Start3DSound( Sound3D* a3DSound, PolarPosition4D* Pos4D )
```

Description

This function starts the DSP instrument associated with a 3D Sound and moves the sound from the center of the head to the position specified with Pos4D. The source sound should be started after a call to Start3DSound so that this movement is not audible.

Arguments

a3DSound

a pointer to a Sound3D structure, previously set using a call to Create3DSound().

Pos4D

a pointer to a PolarPosition4D containing coordinates of the sound's starting position. See *<:audio:sound3d.h>*.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in libmusic.a V30.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:sound3d.h>, libmusic.a, libspmath.a, System.m2/Modules/audio

See Also

Create3DSound(), Delete3DSound(), Stop3DSound(), Move3DSound(),
Move3DSoundTo(), Get3DSoundInstrument(), Get3DSoundPos(),
Get3DSoundParms()

Stop3DSound

Stops a 3D Sound.

Synopsis

```
Err Stop3DSound( Sound3D* a3DSound )
```

Description

This function stops the DSP instrument associated with the sound. If stopped before the source sound, this can result in an audible click, so the function should be called after the source sound has been stopped.

Arguments

`a3DSound`
a pointer to a `Sound3D` structure, previously set using a call to `Create3DSound()`.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs.

Implementation

Library call implemented in `libmusic.a` V30.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:sound3d.h>`, `libmusic.a`, `libspmath.a`, `System.m2/Modules/audio`

See Also

`Create3DSound()`, `Delete3DSound()`, `Start3DSound()`, `Move3DSound()`,
`Move3DSoundTo()`, `Get3DSoundInstrument()`, `Get3DSoundPos()`,
`Get3DSoundParms()`

spAddMarker

Add a new SPMarker to an SPSound.

Synopsis

```
Err spAddMarker (SPSound *sound, uint32 position,
                const char *markerName)
```

Description

Adds a new SPMarker to the specified SPSound.

All markers added to an SPSound are automatically freed when the SPSounds is freed by spRemoveSound(). A marker may be manually freed by calling spRemoveMarker().

Arguments

sound

Pointer to an SPSound to which to add an SPMarker.

position

Position (in frames) within the sound data for the new marker. The range is 0..nframes, where nframes is the length of the sound in frames. 0 places the new marker before the first frame in the sound; nframes places the marker after the last frame in the sound. This position must be byte-aligned, or else this function returns ML_ERR_BAD_SAMPLE_ALIGNMENT.

markerName

Name for the new marker. This name must be unique for the sound. Names are compared case-insensitively. The name passed in to this function is copied, so the caller need not keep the string buffer pointed to by markerName intact after calling this function.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

```
spRemoveMarker(),      spBranchAtMarker(),      spStopAtMarker(),
spContinueAtMarker(),  spSetMarkerDecisionFunction(),
spClearMarkerDecisionFunction()
```

spAddSample

Create an SPSSound for a Sample Item.

Synopsis

```
Err spAddSample (SPSSound **resultSound, SPPlayer *player, Item
sample)
```

Description

Creates an SPSSound for the specified Sample Item and adds it to the specified player. This is useful for playing back sounds that are already in memory.

This function queries the sample item for its properties (e.g. number of channels, size of frame, number of frames, loop points, etc). The sound is checked for sample frame formatting compatibility with the other SPSSounds in the SPPlayer and for buffer size compatibility. A mismatch causes an error to be returned.

Once that is done, the following special SPMarkers are created for the new SPSSound:

SP_MARKER_NAME_BEGIN

Set to the beginning of the sample.

SP_MARKER_NAME_END

Set to the end of the sample.

SP_MARKER_NAME_SUSTAIN_BEGIN

Set to the beginning of the sustain loop if the sample has a sustain loop.

SP_MARKER_NAME_SUSTAIN_END

Set to the end of the sustain loop if the sample has a sustain loop.

SP_MARKER_NAME_RELEASE_BEGIN

Set to the beginning of the release loop if the sample has a release loop.

SP_MARKER_NAME_RELEASE_END

Set to the end of the release loop if the sample has a release loop.

Since a Sample Item has no provision for storing any markers other than the loop points, an SPSSound created from a sample item returned by LoadSample() will not have any of the markers from the AIFF file other than the ones listed above.

The length of the sample and all of its loop points must be byte-aligned or else this function will return ML_ERR_BAD_SAMPLE_ALIGNMENT.

All SPSSounds added to an SPPlayer are automatically disposed of when the SPPlayer is deleted with spDeletePlayer() (by calling spRemoveSound()).

You can manually dispose of an SPSound with `spRemoveSound()`.

Arguments

`resultSound`

Pointer to buffer to write resulting SPSound pointer. Must be supplied or else this function returns `ML_ERR_BADPTR`.

`player`

Pointer to an SPPlayer.

`sample`

Item number of a sample to add.

Return Value

Non-negative value on success; negative error code on failure.

Outputs

A pointer to an allocated SPSound is written to the buffer pointed to by `resultSound` on success. `NULL` is written to this buffer on failure.

Notes

Since all SPSounds belonging to an SPPlayer are played by the same sample player instrument, they must all have the same frame sample frame characteristics (width, number of channels, compression type, and compression ratio).

SPSound to SPSound cross verification is done: an error is returned if they don't match. However, there is no way to verify the correctness of sample frame characteristics for the instrument supplied to `spCreatePlayer()`.

Caveats

There is no restriction on adding Sample Item class SPSounds while the SPPlayer is playing, but removing them while playing can be dangerous. See the Caveats section of `spRemoveSound()` for more details on this.

Note that this only applies to Sample Item class SPSounds (the kind made with this function). There is no restriction on adding _or_ removing AIFF Sound File class SPSounds while playing.

The sound player will not work correctly unless the sample frame size for the sample follows these rules:

If frame size < 1 byte, then frames must not span byte boundaries. There must be an integral number of frames per byte.

If frame size >= 1, then all frame boundaries must fall on byte boundaries. There must be an integral number of bytes per frame.

Implementation

Library call implemented in `libmusic.a V24`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`spRemoveSound()`, `spAddSoundFile()`

spAddSoundFile

Create an SPSound for an AIFF sound file.

Synopsis

```
Err spAddSoundFile (SPSound **resultSound, SPPlayer *player,  
                    const char *fileName)
```

Description

Creates an SPSound for the specified AIFF sound file and adds it to the specified player. SPSounds created this way cause the player to spool the sound data directly off of disc instead of buffering the whole sound in memory. This is useful for playing back really long sounds.

This function opens the specified AIFF file and scans collects its properties (e.g. number of channels, size of frame, number of frames, markers, etc). The sound is checked for sample frame formatting compatibility with the other SPSounds in the SPPlayer and for buffer size compatibility. A mismatch causes an error to be returned.

Once that is done, all of the markers from the AIFF file are translated into SPMarkers. Additionally the following special markers are created:

SP_MARKER_NAME_BEGIN

Set to the beginning of the sound data.

SP_MARKER_NAME_END

Set to the end of the sound data.

SP_MARKER_NAME_SUSTAIN_BEGIN

Set to the beginning of the sustain loop if the sound file has a sustain loop.

SP_MARKER_NAME_SUSTAIN_END

Set to the end of the sustain loop if the sound file has a sustain loop.

SP_MARKER_NAME_RELEASE_BEGIN

Set to the beginning of the release loop if the sound file has a release loop.

SP_MARKER_NAME_RELEASE_END

Set to the end of the release loop if the sound file has a release loop.

The file is left open for the entire life of this type of SPSound for later reading by the player.

The length of the sound file and all of its markers must be byte-aligned or else this function will return ML_ERR_BAD_SAMPLE_ALIGNMENT.

All SPSounds added to an SPPlayer are automatically disposed of when the SPPlayer is deleted with `spDeletePlayer()` (by calling `spRemoveSound()`). You can manually dispose of an SPSound with `spRemoveSound()`.

Arguments

`resultSound`

Pointer to buffer to write resulting SPSound pointer. Must be supplied or else this function returns `ML_ERR_BADPTR`.

`player`

Pointer to an SPPlayer.

`fileName`

Name of an AIFF file to read.

Return Value

Non-negative value on success; negative error code on failure.

Outputs

A pointer to an allocated SPSound is written to the buffer pointed to by `resultSound` on success. NULL is written to this buffer on failure.

Implementation

Library call implemented in `libmusic.a V24`.

Notes

SoundDesigner II has several classes of markers that it supports in sound: loop, numeric, and text. When it saves to an AIFF file, it silently throws away all but the first 2 loops that may be in the edited sound. Numeric markers are written to an AIFF file with a leading "# " which SDII apparently uses to recognize numeric markers when reading an AIFF file. It unfortunately ignores the rest of the marker name in that case, making the actual numbers somewhat variable. Text markers, thankfully, have user editable names that are saved verbatim in an AIFF file. For this reason, we recommend using only text markers (and possibly loops 1 and 2) when preparing AIFF files for use with the sound player in SoundDesigner II.

Since all SPSounds belonging to an SPPlayer are played by the same sample player instrument, they must all have the same frame sample frame characteristics (width, number of channels, compression type, and compression ratio).

SPSound to SPSound cross verification is done: an error is returned if they don't match. However, there is no way to verify the correctness of sample frame characteristics for the instrument supplied to `spCreatePlayer()`.

Caveats

The sound player will not work correctly unless the sample frame size for the sample follows these rules:

If frame size < 1 byte, then frames must not span byte boundaries. There must be an integral number of frames per byte.

If frame size >= 1, then all frame boundaries must fall on byte boundaries. There must be an integral number of bytes per frame.

Associated Files

<:audio:soundplayer.h>, libmusic.a, System.m2/Modules/iff

See Also

spRemoveSound(), spAddSample()

spBranchAtMarker

Set up a static branch at a marker.

Synopsis

```
Err spBranchAtMarker (SPSound *fromSound, const char
*fromMarkerName,
                    SPSound *toSound, const char *toMarkerName)
```

Description

Sets up a static branch from one marker to another. Barring the result of a marker or default decision function, when the playback encounters the 'from' marker, the playback will skip to the 'to' marker seamlessly.

Markers can have one of 3 static actions: continue, branch, or stop. This action can be changed at any time, including while the player is playing.

Arguments

fromSound

Pointer to SPSound containing marker to branch from.

fromMarkerName

Name of marker in fromSound to branch from. All markers except markers at the beginning of sounds are considered valid "from" markers.

toSound

Pointer to SPSound containing marker to branch to.

toMarkerName

Name of marker branch to in toSound. All markers except markers at the end of sounds are considered valid "to" markers.

Return Value

Non-negative value on success; negative error code on failure.

Notes

Both SPSounds must belong to the same SPPlayer.

Good sounding results require good placement of markers in the sounds.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spStopAtMarker(), spContinueAtMarker(),
spSetMarkerDecisionFunction(),

`spSetDefaultDecisionFunction()`

spClearDefaultDecisionFunction Clears global decision function.

Synopsis

```
Err spClearDefaultDecisionFunction (SPPlayer *player)
```

Description

Removes a global decision function installed by `spSetDefaultDecisionFunction()`.

Arguments

player

SPPlayer from which to remove default decision function.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Macro implemented in `<:audio:soundplayer.h>` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spSetDefaultDecisionFunction()`

spClearMarkerDecisionFunction Clears a marker decision function.

Synopsis

```
Err spClearMarkerDecisionFunction (SPSound *sound,  
                                   const char *markerName)
```

Description

Removes a marker decision function installed by `spSetMarkerDecisionFunction()`.

Arguments

`sound`
Pointer to `SPSound` containing marker from which to remove decision function.

`markerName`
Name of marker in sound from which to remove decision function.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Macro implemented in `<:audio:soundplayer.h>` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spSetMarkerDecisionFunction()`

spContinueAtMarker

Clear static branch at a marker.

Synopsis

```
Err spContinueAtMarker (SPSound *sound, const char *markerName)
```

Description

Restores the marker to its default action of playing through (for a marker in the middle of a sound), or stopping (for a marker at the end of a sound), barring the results from a marker or default decision function. This resets the effect of `spBranchAtMarker()` or `spStopAtMarker()`.

Markers can have one of 3 static actions: continue, branch, or stop. This action can be changed at any time, including while the player is playing.

Arguments

`sound`

Pointer to `SPSound` containing marker to clear branch from.

`markerName`

Name of marker in sound to clear branch from. All markers except markers at the beginning of sounds are considered valid for this function.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in `libmusic.a` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spBranchAtMarker()`, `spStopAtMarker()`, `spSetMarkerDecisionFunction()`, `spSetDefaultDecisionFunction()`

spCreatePlayer

Create an SPPlayer.

Synopsis

```
Err spCreatePlayer (SPPlayer **resultPlayer,  
                    Item samplerInstrument,  
                    uint32 numBuffers, uint32 bufferSize,  
                    void * const customBuffers[])
```

Description

Creates an SPPlayer (the top-level player context). The player needs:

- a sample player DSP instrument to play the sound.
- a set of identically-sized audio buffers used to spool sound off of disc.

The client must supply the sample player instrument. Buffers can be allocated automatically or supplied by the client.

At least 2 buffers are required in order to permit smooth sound playback. Since each buffer will have a signal allocated for it, there can be no more than 23 buffers (the number of free signals for a task that has no signals allocated). 4 or 5 is probably a comfortable number of buffers.

Buffer size must be at least 512 bytes for in-memory sounds or 2 * blocksize + 514 for sounds off disc. Typically much larger buffers are required to smoothly playback sounds directly off disk (e.g. 16K or so for 44100 frame/sec playback of a 16-bit monophonic sound).

For disc-based sounds, the buffer size, truncated to a multiple of block size, represents the maximum amount of data that can be read from the disc in a single I/O request. Given the nature of data being read from a CD, the larger the buffer, the more efficiently the player will read data off of disc. In order to guarantee smooth sound playback, there must be enough data spooled at all times to cover the read of 1 buffer. Obviously, this requires that the data can be read faster than it can be played.

The total size of all of the buffers represents the maximum latency between reading and the actual sound output (the responsiveness to branches and decisions). The maximum latency can be computed as follows:

$$\text{max latency (sec)} = \text{numBuffers} * \text{bufferSize} / \text{frame size} / \text{sample rate}$$

For example:

```
numBuffers = 4
```

```
bufferSize = 16384 bytes
```

frame size = 2 (16-bit monophonic)

sample rate = 44100 frames / sec

max latency = $4 * 16384 / 2 / 44100 = 0.743$ sec

With the above buffer configuration and playback rate, the results of branches and decision functions will be heard no later than about 3/4 second afterwards.

The client is left with the problem of striking a good balance between efficient I/O usage and responsiveness for any specific application.

Use `spDeletePlayer()` to dispose of this `SPPlayer` when you are done with it.

Arguments

`resultPlayer`

Pointer to buffer to write resulting `SPPlayer` pointer. Must be supplied or else this function returns `ML_ERR_BADPTR`.

`samplerInstrument`

Item number of sample player instrument to be used to play back sounds with this `SPPlayer`. You may use any instrument, including a patch, as long as it has exactly one input FIFO and no output FIFOs. All sounds added to the player must be of a format that this sampler instrument can play. The player has no way to verify correctness of instrument, however. You are responsible for all connections to this instrument (e.g., connecting to `line_out.dsp` in order to hear the output).

`numBuffers`

Number of buffers to use. Valid range is 2..23 (limited by available signals).

`bufferSize`

Size of each buffer in bytes. For in-memory sounds, must be at least 512 bytes. For sounds to spool directly off disc, must be at least $2 * \text{blocksize} + 514$. In practice larger buffers for disc-based sounds are probably required in order to provide smooth sound playback.

`customBuffers`

Table of pointers to client-supplied buffers. If `NULL` is passed in for this argument, the player will allocate memory for the specified buffers.

Use this if you want to allocate your own buffers instead of letting the player do it. The table must contain `numBuffers` pointers to unique buffers that must be `bufferSize` in length.

The customBuffers table (if there is one) is copied into the SPPlayer structure, so the table need not remain valid after this function returns.

Return Value

Non-negative value on success; negative error code on failure.

Outputs

A pointer to an allocated SPPlayer is written to the buffer pointed to by resultPlayer on success. NULL is written to this buffer on failure.

Caveats

The sound player allocates one signal from your task for each buffer. Keep this in mind when deciding how many buffers to use.

The sound player will not work correctly unless the sample frame size for the sample follows these rules:

If frame size < 1 byte, then frames must not span byte boundaries. There must be an integral number of frames per byte.

If frame size >= 1, then all frame boundaries must fall on byte boundaries. There must be an integral number of bytes per frame.

Implementation

Library call implemented in libmusic.a V24.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundplayer.h>, libmusic.a, System.m2/Modules/audio

See Also

spDeletePlayer(), spAddSoundFile(), spAddSample(), spStartReading(), spStartPlaying(), spStop()

SPDecisionFunction

Typedef for decision callback functions

Synopsis

```
typedef Err (*SPDecisionFunction) (SPAction *resultAction,  
                                   void *decisionData,  
                                   SPSSound *sound,  
                                   const char *markerName)
```

Description

Client-supplied callback functions of this form may be installed in each marker and/or as a default for an SPPlayer.

A marker's action is processed when the sound data for that marker's position is read by the player when any of the following is true:

- there is a default decision function for the marker's SPPlayer.
- the marker has a marker decision function.
- the marker specifies a non-trivial action (branch or stop).
- the marker is at the end of a sound.

Normally, disc I/O is performed based on the size of each buffer. When one of the above conditions is true for a marker, the player will split up disc I/O at that marker in order to prepare for a possible branch. This impacts disc I/O efficiency. It is important to avoid having so many markers with non-trivial actions as to cripple the player's ability to produce smooth sounding output. Since default decision functions affect every marker, it is necessary to insure that the marker spacing is such that smooth playback can still be produced. This is unfortunately one of those things that requires a bit of trial and error.

The following steps are used in determining what action the SPPlayer will take when processing a marker's action:

- If the marker has a decision function, call it. If the marker's decision function sets its resultAction (e.g. by a call to spSetBranchAction() or spSetStopAction()), then take that action.
- Otherwise, if there is a default decision function, call it. If the default decision function sets its resultAction (e.g. by a call to spSetBranchAction() or spSetStopAction()), then take that action.
- Otherwise, if the marker specifies some static action (branch or stop), take that action.
- Otherwise, if the marker is at the end of a sound, stop reading.
- Otherwise continue reading the sound after the marker.

A decision function MAY do almost anything to the SPPlayer that owns this marker including adding new sounds or markers, changing the static action for this or any other marker, changing the default or marker decision functions for this or any other marker, deleting this or any other marker or sound (with the below caveats in mind).

A decision function MUST NOT do any of the following:

- call any function that changes the player state (e.g., `spDeletePlayer()`, `spStop()`, `spStartReading()`, `spStartPlaying()`, `spService()`, etc) for the current SPMarker's SPPlayer.
- delete the current SPMarker's SPSound since this has the side effect of calling `spStop()`.
- delete the current marker without setting up `resultAction` to stop or branch to another marker.
- take a long time to execute.

Arguments

`resultAction`

SPAction to optionally be filled out by decision function. If it isn't filled out, the sound player ignores the call to the decision function and continues as if the decision function hadn't been installed.

`decisionData`

Pointer to client data passed to
`spSetMarkerDecisionFunction()` or
`spSetDefaultDecisionFunction()`.

`sound`

Pointer to SPSound containing marker for which the player is requesting a playback decision.

`markerName`

Name of marker for which the player is requesting a playback decision.

Return Value

Client should return a non-negative value for success, or a negative error code on failure. An error code returned to the sound player causes the sound player to stop reading and return that error code back to the caller.

Outputs

Decision function can set the SPAction by calling one of the following:

`spSetBranchAction()`

Specifies a marker to branch to as the result of this decision function.

`spSetStopAction()`

Specifies that reader should stop as the result of this decision function.

If the decision does not set the `SPAction`, the sound player ignores the the decision function (acts as if decision function hadn't been installed).

Examples

```
Err mydecision (SPAction *resultAction, int32 *remaining,
               SPSound *sound, const char *markerName)
{
    // stop when remaining count reaches zero
    if ((*remaining)-- <= 0) {
        return spSetStopAction (resultAction);
    }

    // do nothing (take default action) otherwise
    return 0;
}
```

Implementation

Typedef for callback function defined in `<:audio:soundplayer.h>` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spSetMarkerDecisionFunction()`, `spSetDefaultDecisionFunction()`,
`spSetBranchAction()`, `spSetStopAction()`

spDeletePlayer

Delete an SPPlayer.

Synopsis

```
Err spDeletePlayer (SPPlayer *player)
```

Description

Deletes an SPPlayer created by `spCreatePlayer()`. Automatically stops the SPPlayer and calls `spRemoveSound()` for all SPSounds belonging to the SPPlayer. If the `spCreatePlayer()` allocated buffers, those buffers are automatically freed. Custom allocated buffers passed in to `spCreatePlayer()` are not deleted.

Arguments

`player`
Pointer to an SPPlayer to delete. Can be NULL.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in `libmusic.a` V24.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

```
spCreatePlayer(), spRemoveSound(), spRemoveMarker()
```

spDumpPlayer

Print debug information for sound
player

Synopsis

```
void spDumpPlayer (const SPPlayer *player)
```

Description

Prints out a bunch of debug information for an SPPlayer including:

- list of the SPPlayer's SPSounds
- list of each SPSound's SPMarkers
- cross reference of static branches between markers

Arguments

player
 Pointer to SPPlayer to print.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spCreatePlayer()

spFindMarkerName

Return pointer an SPMarker by looking up its name.

Synopsis

```
SPMarker *spFindMarkerName (const SPSound *sound, const char  
*markerName)
```

Description

Locates the specified marker name in the specified sound.

Arguments

`sound`
Pointer to SPSound to search for marker.

`markerName`
Name of marker in sound to find.

Return Value

Pointer to matched SPMarker on success; NULL on failure.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spAddMarker(), spRemoveMarker(), SPDecisionFunction

spGetMarkerName

Get name of an SPMarker.

Synopsis

```
const char *spGetMarkerName (const SPMarker *marker)
```

Description

Returns a pointer to the name of the SPMarker.

Arguments

marker
Pointer to SPMarker to interrogate.

Return Value

Pointer to the SPMarker name string.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spAddMarker(), spFindMarkerName(), spGetMarkerPosition()

spGetMarkerPosition

Get position of an SPMarker.

Synopsis

```
uint32 spGetMarkerPosition (const SPMarker *marker)
```

Description

Returns frame position of the SPMarker.

Arguments

marker
Pointer to SPMarker to interrogate.

Return Value

Frame position of the SPMarker.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spAddMarker(), spFindMarkerName(), spGetMarkerName()

spGetPlayerFromSound

Get SPPlayer that owns an SPSound.

Synopsis

```
SPPlayer *spGetPlayerFromSound (const SPSound *sound)
```

Description

Returns a pointer to the SPPlayer to which the specified SPSound was added.

Arguments

`sound`
Pointer to SPSound to interrogate.

Return Value

Pointer to SPPlayer that owns this SPSound.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spAddSoundFile(), spAddSample()

spGetPlayerSignalMask

Get set of signals player will send to client.

Synopsis

```
int32 spGetPlayerSignalMask (const SPPlayer *player)
```

Description

Returns the set of signals that the player will send to the client. The client must wait on this signal set between calls to `spService()`. This signal set is constant for the life of the `SPPlayer`. This function cannot fail.

Arguments

`player`
Pointer to `SPPlayer`.

Return Value

Set of signals suitable for passing to `WaitSignal()`.

Implementation

Library call implemented in `libmusic.a` V24.

Associated Files

<:audio:soundplayer.h>, `libmusic.a`

See Also

`spCreatePlayer()`, `spService()`

spGetPlayerStatus

Get current status of an SPPlayer.

Synopsis

```
int32 spGetPlayerStatus (const SPPlayer *player)
```

Description

Returns a set of flags indicating the current state of an SPPlayer. The most useful thing about this function is that it can be used to find out when an SPPlayer has finished playing.

Arguments

```
- - player
      Pointer to SPPlayer to interrogate.
```

Return Value

Any combination of the following SP_STATUS_F_ flags (always a non-negative value):

SP_STATUS_F_BUFFER_ACTIVE

This flag indicates that there's something in the sound buffers waiting to be played. It is set or cleared by spStartReading(), spService(), and cleared by spStop().

When this flag has been cleared after playback has started, all of the sound data has been played: it's safe to stop.

SP_STATUS_F_READING

This flag indicates that there's more data to read. It is set by spStartReading() and cleared when there is no more to read (by spStartReading() or spService()), or when spStop() is called.

SP_STATUS_F_PLAYING

This flag indicates that playback is underway. It is set by spStartPlaying() and cleared by spStop().

SP_STATUS_F_PAUSED

This flag indicates that player has been paused. It is set by spPause() and cleared by spResume(), spStop(), spStartPlaying(). This flag is really only meaningful when SP_STATUS_F_PLAYING is set.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spStartReading(), spStartPlaying(), spStop()

spGetSoundFromMarker

Get SPSSound that owns an SPMarker.

Synopsis

```
SPSSound *spGetSoundFromMarker (const SPMarker *marker)
```

Description

Returns a pointer to the SPSSound to which the specified SPMarker belongs.

Arguments

marker
Pointer to SPMarker to interrogate.

Return Value

Pointer to SPSSound that owns this SPMarker.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spAddMarker(), spFindMarkerName(), spGetMarkerPosition(),
spGetMarkerName()

spIsSoundInUse

Determines if SPPlayer is currently reading from a particular SPSound.

Synopsis

```
bool spIsSoundInUse (const SPSound *sound)
```

Description

Determines if SPPlayer is currently reading from a particular SPSound. An SPSound cannot be removed from its owning SPPlayer while it is being read from while the player is running. This function is necessary for a client to know when it is safe to remove an SPSound.

Arguments

sound
Pointer to SPSound to test.

Return Value

TRUE if sound is currently being read by player; FALSE otherwise.

Caveats

Returns FALSE too early for Sample Item class SPSounds. Since the spooler is given a pointer to the Sample Item's memory directly, rather than copying it into the spooler buffers, it mustn't return FALSE until a Sample Item SPSound has actually finished playing, not finished being read.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spRemoveSound(), spStop()

spLinkSounds

Branch at end of one sound to beginning of another.

Synopsis

```
Err spLinkSounds (SPSound *fromSound, SPSound *toSound)
```

Description

This is a convenient macro for linking multiple sounds together. It simply does:

```
spBranchAtMarker (fromSound,      SP_MARKER_NAME_END,      toSound,
SP_MARKER_NAME_BEGIN)
```

Arguments

fromSound

Pointer to SPSound to link from.

toSound

Pointer to SPSound to link to the end of fromsound.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Macro implemented in `<:audio:soundplayer.h>` V24.

Notes

Both SPSounds must belong to the same SPPlayer.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spBranchAtMarker()`, `spLoopSound()`

spLoopSound

Branch at end of sound back to the beginning.

Synopsis

```
Err spLoopSound (SPSound *sound)
```

Description

This is a convenient macro for looping a single sound. It simply does:

```
spBranchAtMarker      (sound,      SP_MARKER_NAME_END,      sound,  
SP_MARKER_NAME_BEGIN)
```

Arguments

sound
 Pointer to SPSound to loop.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Macro implemented in `<:audio:soundplayer.h>` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spBranchAtMarker()`, `spLinkSounds()`

spPause

Pause an SPPlayer.

Synopsis

```
Err spPause (SPPlayer *player)
```

Description

Pauses playback of an SPPlayer. Use this to stop playback dead in its tracks and be able to pick up at the same spot. This differs from spStop() in that playback cannot be restarted at the position at which spStop() occurred. Resume playback with spResume().

Sets SP_STATUS_F_PAUSED. Calling spPause() multiple times has no effect, the calls do not nest. Calling spPause() while the SPPlayer's SP_STATUS_F_PLAYING flag is not set has no effect. The paused state is superceded by a call to spStop() or spStartPlaying().

There is presently no way to know whether an SPPlayer is paused.

Data can still be spooled while paused if there were any completed buffers prior to pausing.

Arguments

player
 Pointer to SPPlayer to pause.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundplayer.h>, libmusic.a, System.m2/Modules/audio

See Also

spResume(), spStartPlaying(), spStop()

spRemoveMarker

Manually remove an SPMarker from an SPSound.

Synopsis

```
Err spRemoveMarker (SPMarker *marker)
```

Description

Removes and frees the specified marker from the sound that it belongs to. All markers belonging to an SPSound are automatically disposed of when that sound is disposed of with `spRemoveSound()` or when the player that owns that sound is deleted with `spDeletePlayer()`. Use this function if you want to remove a marker manually.

This function cannot be used to remove a permanent marker (`SP_MARKER_NAME_BEGIN` or `SP_MARKER_NAME_END`) from a sound.

Arguments

`marker`

Pointer to an SPMarker to remove. Can be NULL. If this is one of the permanent markers (`SP_MARKER_NAME_BEGIN` or `SP_MARKER_NAME_END`) this function does nothing and returns `ML_ERR_INVALID_MARKER`.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in `libmusic.a` V24.

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`

See Also

`spFindMarkerName()`, `spAddMarker()`, `spRemoveSound()`, `spDeletePlayer()`

spRemoveSound

Manually remove an SPSSound from an SPPlayer.

Synopsis

```
Err spRemoveSound (SPSSound *sound)
```

Description

Removes and frees the specified sound from the SPPlayer that it belongs to. All markers belonging to this sound including any the client may have added, are automatically freed by this function.

All sounds belonging to an SPPlayer are automatically disposed of when that SPPlayer is disposed of with spDeletePlayer(). Use this function if you want to remove a sound manually.

For sound file class SPSSounds, the file opened by spAddSoundFile() is closed. The sample item passed into spAddSample() is left behind after this spRemoveSound() for the client to clean up.

An SPSSound cannot be removed from its owning SPPlayer while it is being read from while the player is running. spIsSoundInUse() can be used to determine if the sound is being read. If this function is called for an SPSSound for which spIsSoundInUse() returns TRUE, the SPPlayer is stopped by calling spStop().

Arguments

sound

Pointer to an SPSSound to remove. Can be NULL.

Return Value

Non-negative value on success; negative error code on failure.

Caveats

This function only stops the SPPlayer when while the SPPlayer is still reading from the SPSSound being removed, which is fine for AIFF Sound File class SPSSounds. For Sample Item class SPSSounds, the player passes pointers to the Sample Item's sound data in memory the SoundSpooler rather than copying the sound data to the SPPlayer's buffers. That means that the sound data must remain valid until the SoundSpooler has completely played the sound data, which happens some amount of time after the SPPlayer is done 'reading' that data.

If you remove a Sample Item class SPSSound during that window of vulnerability and then delete the data for the Sample Item (e.g. UnloadSample()), the SoundSpooler is then pointing to freed memory, whose contents may be clobbered by another task at any time. Unfortunately you may not hear bad results if you do this inadvertently because there's no guarantee that freed memory will get trashed (unless you are using memdebug).

So, avoid removing Sample Item class SPSSounds while the SPPlayer is playing. Or at least go to measures to insure that the Sample Item class

SPSound you are removing is not in the spooler (e.g. it has never been played, or at least not since the last `spStop()`).

Implementation

Library call implemented in `libmusic.a` V24.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`spAddSample()`, `spAddSoundFile()`, `spRemoveMarker()`, `spDeletePlayer()`,
`spIsSoundInUse()`

spResume

Resume playback of an SPPlayer after being paused.

Synopsis

```
Err spResume (SPPlayer *player)
```

Description

Resumes playback of an SPPlayer after being paused.

Clears the SP_STATUS_F_PAUSED flag. Calling spResume() multiple times has no effect, the calls do not nest. Calling spResume() while the SPPlayer is the SP_STATUS_F_PLAYING flag is not set has no effect. The paused state is superceded by a call to spStop() or spStartPlaying().

There is presently no way to know if an SPPlayer is paused.

Arguments

player
 Pointer to SPPlayer to resume.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundplayer.h>, libmusic.a, System.m2/Modules/audio

See Also

spPause(), spStartPlaying(), spStop()

spService

Service SPPlayer

Synopsis

```
int32 spService (SPPlayer *player, int32 signals)
```

Description

This function does the following:

- processes signals received by the client's sent by the SPPlayer.
- reads and spools more sound data, processing marker actions along the way.

This function synchronizes the SPPlayer's SoundSpooler to the state of the signals read from the last WaitSignal(). You must call it after waiting for SPPlayer's signals before calling any other sound player function for this SPPlayer (with the exception of spDeletePlayer(), which can tolerate the spooler being out of sync with the task's signals).

This function can set or clear the SP_STATUS_F_BUFFER_ACTIVE flag, depending on buffer usage. It clears SP_STATUS_F_READING when there's no more sound data to read. This function may be called under any player state including when SP_STATUS_F_PLAYING is not set without any ill effects.

This function must be called exactly once for each WaitSignal() on an SPPlayer's signals. It will almost certainly have dangerous results if fed incorrect signals (signals other than from the most recent WaitSignal(), or signals that it has already processed).

Arguments

player

Pointer to SPPlayer to service.

signals

Signals last received to process. Ignores any signals that do not belong to this SPPlayer. Does nothing if none of the SPPlayer's signals are set.

Return Value

Non-negative player status flags (SP_STATUS_F_) on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Examples

```
// error checking omitted for brevity

{
    const int32 playersigs = spGetPlayerSignalMask (player);
```

```
    // service player until it's done
while (spGetPlayerStatus(player) & SP_STATUS_F_BUFFER_ACTIVE) {
    const int32 sigs = WaitSignal (playersigs |
                                   othersignals1 |
                                   othersignals2);

    // service player before servicing other
    // things that might affect the player
    spService (player, playersigs);

    // service other things
    if (sigs & othersignals1) {
    }
    if (sigs & othersignals2) {
    }
}
}
```

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spGetPlayerSignalMask(), spGetPlayerStatus(), spStartReading(),
spStartPlaying()

spSetBranchAction

Set up an SPAction to branch to the specified marker.

Synopsis

```
Err spSetBranchAction (SPAction *action, SPSound *toSound,  
                      const char *toMarkerName)
```

Description

Sets up the SPAction to cause a branch to the specified marker as the resulting action of a decision function.

Arguments

action

Pointer to an SPAction to fill out. This must be the pointer passed to your decision function.

toSound

Pointer to SPSound containing marker to branch to. This SPSound must belong to the same SPPlayer as the SPMarker for which the decision function was called.

toMarkerName

Name of marker to in toSound branch to. All markers except markers at the end of sounds are considered valid "to" markers.

Return Value

Non-negative value on success; negative error code on failure.

Outputs

On success, fills out SPAction to cause a branch to the specified marker. On failure, nothing is written to the SPAction.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

SPDecisionFunction, spSetStopAction(), spBranchAtMarker()

spSetDefaultDecisionFunction

Install a global decision function to be called for every marker.

Synopsis

```
Err spSetDefaultDecisionFunction (SPPlayer *player,  
                                SPDecisionFunction decisionFunc,  
                                void *decisionData)
```

Description

Install a global decision function that is to be called when the player reaches each marker. This can be used as a way to do some common operation at multiple locations during playback (e.g. processing a script).

Clear the global decision function with `spClearDefaultDecisionFunction()`.

Arguments

player
SPPlayer to which to install default decision function.

decisionFunc
An SPDecisionFunction to install, or NULL to clear.

decisionData
A pointer to client-supplied data to be passed to decisionFunc when called. Can be NULL.

Return Value

Non-negative value on success; negative error code on failure.

Notes

The player normally optimizes reading data from disc by ignoring markers that have neither a branch destination nor a marker decision function. When a default decision function is installed, the player must interrupt reading data from disc in preparation for a potential branch at EVERY marker, even if the default decision function does nothing. This can severely impact performance depending on the placement of markers.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

`spClearDefaultDecisionFunction()`, `spSetMarkerDecisionFunction()`,
`SPDecisionFunction`

spSetMarkerDecisionFunction

Install a marker decision function.

Synopsis

```
Err spSetMarkerDecisionFunction (SPSound *sound,  
                                const char *markerName,  
                                SPDecisionFunction decisionFunc,  
                                void *decisionData)
```

Description

Install a decision function that is to be called when the player reaches the specified marker. Each marker can have a different decision function.

Clear the marker's decision function with `spClearMarkerDecisionFunction()`.

Arguments

sound

Pointer to SPSound containing marker to which to install decision function.

markerName

Name of marker in sound to which to install decision function.

decisionFunc

An SPDecisionFunction to install, or NULL to clear.

decisionData

A pointer to client-supplied data to be passed to decisionFunc when called. Can be NULL.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Notes

While a marker decision function is installed, reading from disc is interrupted every time this marker is encountered. This can have an impact on performance.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

`spClearMarkerDecisionFunction()`, `spSetDefaultDecisionFunction()`,
`SPDecisionFunction`

spSetStopAction

Set up an SPAction to stop reading.

Synopsis

```
Err spSetStopAction (SPAction *resultAction)
```

Description

Sets up the SPAction to stop reading as the resulting action of a decision function.

Arguments

action
Pointer to an SPAction to fill out. This must be the pointer passed to your decision function.

Return Value

Non-negative value on success; negative error code on failure.

Outputs

On success, fills out SPAction to cause a branch to the specified marker. On failure, nothing is written to the SPAction.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

SPDecisionFunction, spSetBranchAction(), spStopAtMarker()

spStartPlaying

Begin emitting sound for an SPPlayer.

Synopsis

```
Err spStartPlaying (SPPlayer *player, const TagArg *startTagList)
```

```
Err spStartPlayingVA (SPPlayer *player, uint32 startTag1, ...)
```

Description

Begins the actual output of spooled sound. This function begins playback immediately making it suitable for being called in sync with some user activity if necessary. `spStartReading()` prefills the spooler buffers. This function should therefore be called some time after calling `spStartReading()`.

Calling this function starts the process of signals arriving which need to be serviced with `spService()`. This function causes the `SP_STATUS_F_PLAYING` flag to be set.

Call `spStop()` to manually stop or wait for playback to finish by checking `spGetPlayerStatus()` after calling `spService()`.

Arguments

player
Pointer to SPPlayer to start playing.

startTagList
Tag list be passed to `StartInstrument()` for sample player instrument. Useful tags include `AF_TAG_AMPLITUDE_FP` and `AF_TAG_SAMPLE_RATE_FP`. See `StartInstrument()` for details.

Return Value

Non-negative value on success; negative error code on failure.

Notes

The only bad thing that will happen if you call this function before calling `spStartReading()` is that you won't have any control over the precise time that playback begins. In that case, playback would start as soon as `spStartReading()` has read a buffer's worth of data. The player also might starved briefly if the 2nd buffer weren't ready before the first one finished.

Multiple calls to `spStartPlaying()` without an intervening stop of some kind (`spStop()` or marker causing a stop) cause stuttered sound output.

A call to `spStartPlaying()` will supercede a previous call to `spPause()`.

Implementation

Library call implemented in `libmusic.a V24`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundplayer.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

spStartReading(), spStop(), spService(), spGetPlayerStatus(),
StartInstrument()

spStartReading

Start SPPlayer reading from an SPSound.

Synopsis

```
Err spStartReading (SPSound *startSound,  
                   const char *startMarkerName)
```

Description

This function begins the process of spooling data for an SPPlayer, processing marker actions along the way. It completely fills the spooler buffers belonging to the SPPlayer in preparation for playback, which can take an unpredictable amount of time. Therefore, the actual function to begin playback, `spStartPlaying()`, is a separate call so that starting the actual sound playback may be synchronized to some user event.

Normally this function is called while the player's `SP_STATUS_F_READING` status flag is cleared (see `spGetPlayerStatus()`), in which case it merely begins reading at the specified location. The `SP_STATUS_F_READING` flag is then set. If the entire sound data to be played fits completely into the buffers, this flag is cleared again before this function returns.

This function can also be called while the `SP_STATUS_F_READING` flag is set in order to force the playback to a different location without waiting for a marker branch or decision function. This abnormal method of relocating sound playback will almost certainly produce unpleasant sound output, but there may be times when it is necessary. Note that this merely causes reading to begin at a new location. It does not flush the spooler. Anything buffered already will continue play.

Arguments

`startSound`

Pointer to SPSound to start reading from.

`startMarkerName`

Name of marker in `startSound` to start reading from.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in `libmusic.a V24`.

Examples

```
// error checking omitted for brevity  
  
{  
    const int32 playersigs = spGetPlayerSignalMask (player);  
  
    // read from beginning of one of the sounds  
    spStartReading (player, sound1, SP_MARKER_NAME_BEGIN);  
  
    // could wait for some event to trigger playback here  
  
    // begin playback  
    spStartPlayingVA (player,  
                     AF_TAG_AMPLITUDE, 0x7fff,  
                     TAG_END);  
}
```

```
        // service player until it's done
    while (spGetPlayerStatus(player) & SP_STATUS_F_BUFFER_ACTIVE) {
        const int32 sigs = WaitSignal (playersigs);

        spService (player, playersigs);
    }
}
```

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spStartPlaying(), spStop(), spService(), spGetPlayerStatus()

spStop

Stops an SPPlayer.

Synopsis

```
Err spStop (SPPlayer *player)
```

Description

Stops reading and sound playback.

Clears all SP_STATUS_F_ flags. Can be called regardless of SPPlayer's current state. Multiple calls have no effect.

Arguments

player
Pointer to SPPlayer to stop.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:soundplayer.h>, libmusic.a, System.m2/Modules/audio

See Also

```
spStartReading(), spStartPlaying(), spGetPlayerStatus()
```

spStopAtMarker

Stop when playback reaches marker.

Synopsis

```
Err spStopAtMarker (SPSound *sound, const char *markerName)
```

Description

Barring results of a marker or default decision function, causes playback to stop when the player reaches this marker.

Markers can have one of 3 static actions: continue, branch, or stop. This action can be changed at any time, including while the player is playing.

Arguments

sound

Pointer to SPSound containing marker to stop at.

markerName

Name of marker in sound to stop at. All markers except markers at the beginning of sounds are considered valid places to stop.

Return Value

Non-negative value on success; negative error code on failure.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundplayer.h>, libmusic.a

See Also

spBranchAtMarker(), spContinueAtMarker(),
spSetMarkerDecisionFunction(), spSetDefaultDecisionFunction()

SoundBufferFunc

SoundSpooler callback function typedef.

Synopsis

```
typedef int32 (*SoundBufferFunc) ( SoundSpooler *sspl,  
                                   SoundBufferNode *sbn,  
                                   int32 msg )
```

Description

This callback system is a superset of the UserBufferProcessor system. It offers 2 basic improvements over the former:

1. It provides a method of getting buffer start as well as completion notification.
2. UserBufferProcessor can't be called by functions such as `ssplDetachInstrument()` or `ssplPlayData()` because these functions do not have a UserBufferProcessor argument. The SoundBufferFunc is stored in the SoundSpooler structure, so all support functions can call it.

The client can install a callback function pointer of this type in the SoundSpooler by calling `ssplSetSoundBufferFunc()`. Several sound spooler functions send notification messages to this function when the state of a SoundBufferNode changes. When a buffer is started, one of the start class messages is sent to the SoundBufferFunc. When a buffer ends, one of the end class messages is sent.

This callback function is called with a message id, a pointer to the SoundSpooler and SoundBufferNode whose state has changed. The callback function can call non-destructive sound spooler functions like `ssplGetUserData()`, `ssplGetSequenceNum()`, etc, but is not permitted to do anything to change the state of the sound spooler. In particular, calling `ssplRequestBuffer()` or `ssplSendBuffer()` typically will confuse the sound spooler's list processing functions. Also, the state of `SSPL_STATUS_F_ACTIVE` is undefined when inside a SoundBufferFunc function.

Generally, the callback function must be able to handle all message types. To simplify message handling, you can get the class of a message by calling `ssplGetSBMsgClass()`. This is extremely useful if you only care to know that a buffer started or finished, but not the specifics of how it did this.

The callback function is required to return a result. The sound spooler considers a return value ≥ 0 from the callback function to indicate success. This value does NOT propagate back to the client.

If the callback returns a value < 0 , that value is returned to the client through the sound spooler function that called the callback function. In this case, the sound spooler function terminates immediately. Any function that can process multiple SoundBufferNodes (e.g. `ssplAbort()`, or `ssplProcessSignals()`) doesn't complete its active queue processing when the callback function returns an error code. If the cause of the callback failure can be resolved by the client, the client can usually call the offending function again to continue where it left off.

Use of the SoundBufferFunc and UserBufferProcessor systems are mutually exclusive. When a UserBufferProcessor is supplied to a function that supports it and a SoundBufferFunc is installed, that function fails and returns an error code prior to doing anything.

Arguments

`sspl`
SoundSpooler.

sbm
SoundBufferNode changing state.

msg
A message id (SSPL_SBMSG_...). See below.

Messages

Start Class

SSPL_SBMSG_INITIAL_START

Initial Start. `ssplStartSpoolerTags()` sends this message for the first buffer in active queue.

SSPL_SBMSG_LINK_START

Link Start. `ssplProcessSignals()` sends this message for the next buffer in the active queue after it removes a completed buffer.

SSPL_SBMSG_STARVATION_START

Starvation Start. `ssplSendBuffer()` sends this message for the buffer sent to it if that buffer causes the spooler to restart after being starved.

End Class

SSPL_SBMSG_COMPLETE

Complete. `ssplProcessSignals()` sends this message for every completed buffer that it removes from the active queue.

SSPL_SBMSG_ABORT

Abort. `ssplAbort()` sends this message for every buffer (completed or otherwise) that it removes from the active queue.

Return Value

Client should return non-negative value on success, or a negative 3DO error code on failure.

Implementation

Callback function typedef used by libmusic.a V22.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

`ssplSetSoundBufferFunc()`, `ssplGetSBMsgClass()`, `ssplStartSpoolerTags()`, `ssplAbort()`, `ssplProcessSignals()`, `ssplSendBuffer()`

ssplAbort

Aborts SoundSpooler buffers in active queue.

Synopsis

```
Err ssplAbort (
    SoundSpooler *sspl,
    void (*UserBufferProcessor) (SoundSpooler *, SoundBufferNode *)
)
```

Description

ssplAbort() stops the spooler (by calling ssplStopSpooler()) and removes all buffers in the active queue (by calling ssplReset()), including the one currently being played.

If a SoundBufferFunc is installed, a SSPL_SBMSG_ABORT message is sent to it with each SoundBufferNode removed from the active queue. If UserBufferProcessor is non-NULL, each SoundBufferNode removed from the active queue is passed to it. If both a SoundBufferFunc and UserBufferProcessor are supplied, ssplAbort() fails and returns ML_ERR_BAD_ARG.

If SoundBufferFunc fails, ssplAbort() fails immediately and returns the error code returned by SoundBufferFunc without removing the rest of the buffers from the active queue. Calling ssplAbort() again safely picks up where it left off.

Clears SSPL_STATUS_F_STARED, SSPL_STATUS_F_PAUSED, and SSPL_STATUS_F_ACTIVE.

Arguments

sspl

SoundSpooler to abort.

UserBufferProcessor

Pointer to a function to call with each SoundBufferNode removed from the active queue, or NULL.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplStopSpooler(), ssplSendBuffer(), ssplProcessSignals(), ssplReset(), SoundBufferFunc, UserBufferProcessor, ssplGetSpoolerStatus()

ssplAttachInstrument

Attaches new sample player instrument to SoundSpooler.

Synopsis

```
Err ssplAttachInstrument ( SoundSpooler *sspl, Item SamplerIns )
```

Description

`ssplAttachInstrument()` installs a new sample player instrument in the SoundSpooler. Any previously attached sample player instrument is detached (by calling `ssplDetachInstrument()`) before the new one is attached.

Arguments

`sspl`
pointer to a SoundSpooler structure.

`SamplerIns`
Instrument to attach buffer samples to.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Note

Any buffers in the active queue are aborted by calling `ssplAbort()`. Note that there is no provision to supply a `UserBufferProcessor` function to `ssplAttachInstrument()`. An installed `SoundBufferFunc` works correctly, however.

Caveats

Leaves the SoundSpooler in an indeterminate state if this function fails. A subsequent successful call to this function or to `ssplDetachInstrument()` restores the SoundSpooler to sanity.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, libmusic.a, System.m2/Modules/audio

See Also

`ssplDetachInstrument()`, `ssplAbort()`

ssplCreateSoundSpooler

Creates a SoundSpooler.

Synopsis

```
SoundSpooler *ssplCreateSoundSpooler ( int32 NumBuffers, Item
SamplerIns )
```

Description

`ssplCreateSoundSpooler()` creates a SoundSpooler data structure and initializes it. Call `ssplCreateSoundSpooler()` first to create a SoundSpooler data structure.

This function allocates the requested number of SoundBufferNodes for the SoundSpooler. SoundBufferNodes simply carry pointers to sample data; they do not have any embedded sample data. The pointer and length of each sample buffer is stored into a SoundBufferNode with `ssplSetBufferAddressLength()` prior to enqueueing it with `ssplSendBuffer()`.

Arguments

NumBuffers

Specifies the number of buffers. For double buffering of audio data, use NumBuffers=2. The more buffers you use, the less chance you will have of running out of data. A safe number to use is 4-8.

SamplerIns

The sampler Instrument Item that will play the data. For example, you could use a `sampler_16_f2.dsp` instrument to spool 44.1 kHz, 16-bit, stereo sound data. You can call `GetAIFFSampleInfo()` and `SampleFormatToInsName()` to determine the best standard instrument to use for a given sample format. You may use any instrument, including a patch, as long as it has exactly one input FIFO, no output FIFOs, and can play the format of data being spooled. You are responsible for all connections to this instrument (e.g., connecting to `line_out.dsp` in order to hear the output).

This setting can be changed after creation by using `ssplAttachInstrument()`. Also, this can be 0, if you call `ssplAttachInstrument()` before actually starting the spooler.

Return Value

Pointer to new SoundSpooler on success, or NULL on failure.

Implementation

Library call implemented in libmusic.a V21.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

```
ssplDeleteSoundSpooler(), ssplStartSpoolerTags(), ssplSendBuffer(),
ssplProcessSignals()
```

ssplDeleteSoundSpooler

Deletes a SoundSpooler.

Synopsis

```
Err ssplDeleteSoundSpooler ( SoundSpooler *sspl )
```

Description

`ssplDeleteSoundSpooler()` disposes of a `SoundSpooler` created by `ssplCreateSoundSpooler()` after first stopping it with `ssplStopSpooler()`. All `SoundBufferNodes` associated with the `SoundSpooler` are deleted.

Arguments

`sspl`
Pointer to a `SoundSpooler` structure. Starting with `libmusic.a v22`, can be `NULL`.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in `libmusic.a V21`.

Caveats

Doesn't tolerate a `NULL` `sspl` pointer prior to `libmusic.a v22`.

Note

This function does no notification of `SoundBufferNode` removal from the active queue. If this is an issue, call `ssplAbort()` prior to calling this function.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:soundspooler.h>, `libmusic.a`, `System.m2/Modules/audio`

See Also

```
ssplCreateSoundSpooler(), ssplStopSpooler()
```

ssplDetachInstrument

Detaches the current sample player instrument from the SoundSpooler.

Synopsis

```
Err ssplDetachInstrument ( SoundSpooler *sspl )
```

Description

`ssplDetachInstrument()` removes the current sample player instrument from the SoundSpooler. Any buffers in the active queue are aborted (calls `ssplAbort()`). Spooler playback cannot be started until a new instrument is attached with `ssplAttachInstrument()`. Incidentally, `ssplAttachInstrument()` calls this function prior to attaching a new instrument, so it's unlikely that you actually need to call this function directly.

Arguments

`sspl`
Pointer to a SoundSpooler structure.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in `libmusic.a` V21.

Note

Any buffers in the active queue are aborted by calling `ssplAbort()`. Note that there is no provision to supply a `UserBufferProcessor` function to `ssplAttachInstrument()`. An installed `SoundBufferFunc` works correctly, however.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ssplAttachInstrument()`, `ssplAbort()`

ssplDumpSoundBufferNode

Print debug info for SoundBufferNode.

Synopsis

```
void ssplDumpSoundBufferNode ( const SoundBufferNode *sbn )
```

Description

Prints debugging information for a SoundBufferNode.

Arguments

sbn
SoundBufferNode to print.

Implementation

Library call implemented in libmusic.a V21.

Module Open Requirements

```
OpenAudioFolio()
```

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

```
ssplDumpSoundSpooler()
```


ssplDumpSoundSpooler

Print debug info for SoundSpooler.

Synopsis

```
void ssplDumpSoundSpooler ( const SoundSpooler *sspl )
```

Description

Prints debugging information for SoundSpooler including the contents of both the active and free queues (calls `ssplDumpSoundBufferNode()` for each `SoundBufferNode`).

Arguments

`sspl`
SoundSpooler to print.

Implementation

Library call implemented in `libmusic.a V21`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ssplDumpSoundBufferNode()`

ssplGetSBMsgClass

Get class of message passed to `SoundBufferFunc`.

Synopsis

```
int32 ssplGetSBMsgClass ( int32 msg )
```

Description

Returns the class of a message passed to `SoundBufferFunc`.

Arguments

`msg`
Message id passed to `SoundBufferFunc`.

Return Value

One of the `SSPL_SBMMSGCLASS_...` values defined in `<:audio:soundspooler.h>`.

Implementation

Macro implemented in `<:audio:soundspooler.h>` V22.

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`

See Also

`SoundBufferFunc`

ssplGetSequenceNum

Gets Sequence Number of a SoundBufferNode.

Synopsis

```
int32 ssplGetSequenceNum ( SoundSpooler *sspl, SoundBufferNode *sbn
)
```

Description

`ssplGetSequenceNum()` returns the "sequence number" of the specified `SoundBufferNode`. This is an integer that indicates the allocation sequence of the `SoundBufferNode`.

Arguments

`sspl`
Pointer to a `SoundSpooler` structure.

`sbn`
Pointer to a sound buffer node.

Return Value

An integer that is the index of the buffer as allocated by `ssplCreateSoundSpooler()`. It goes from zero to `NumBuffers-1`.

Implementation

Library call implemented in `libmusic.a V21`.

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`

See Also

`ssplCreateSoundSpooler()`

ssplGetSpoolerStatus

Gets SoundSpooler status flags.

Synopsis

```
int32 ssplGetSpoolerStatus (const SoundSpooler *sspl)
```

Description

Returns a set of flags SSPL_STATUS_F_ flags describing the current state of the SoundSpooler.

Arguments

sspl
Pointer to a SoundSpooler structure.

Return Value

Any combination of these flags (always a positive value):

SSPL_STATUS_F_ACTIVE

This flag are SoundBufferNodes in the SoundSpooler's active queue. It is set by `ssplSendBuffer()`. It can be cleared by `ssplProcessSignals()` when all of the buffers in the active queue have completed. It is always cleared by `ssplReset()` and `ssplAbort()`.

You can use this flag as an indicator that all of the sound data submitted by `ssplSendBuffer()` has been played as of the most recent call to `ssplProcessSignals()`.

SSPL_STATUS_F_STARTED

This flag indicates that the SoundSpooler has been started. Unless it has been paused, it is playing sound or will as soon as there is some to play. This flag is set by `ssplStartSpoolerTags()` and cleared by `ssplStopSpooler()` and `ssplAbort()`.

SSPL_STATUS_F_PAUSED

This flag indicates that the SoundSpooler has been paused. It is only meaningful if SSPL_STATUS_F_STARTED is also set. This flag is set by `ssplPause()` and cleared by `ssplResume()`, `ssplStartSpoolerTags()`, and `ssplStopSpooler()`.

Caveats

This state of SSPL_STATUS_F_ACTIVE is undefined when inside any SoundSpooler callback function. This is the case because the callback functions can be called inside the loop that processes the active queue. No guarantee is made about whether a SoundBufferNode is removed from the active queue before or after the callback functions are called.

Implementation

Library call implemented in libmusic.a V24.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

`ssplIsSpoolerActive()`, `ssplSendBuffer()`, `ssplProcessSignals()`

ssplGetUserData

Gets User Data from a SoundBufferNode.

Synopsis

```
void *ssplGetUserData ( SoundSpooler *sspl, SoundBufferNode *sbn )
```

Description

ssplGetUserData() returns the pointer stored in a SoundBufferNode by ssplSetUserData().

Arguments

sspl

Pointer to a SoundSpooler structure.

sbn

Pointer to a sound buffer node.

Return Value

User data pointer stored in a SoundBufferNode by ssplSetUserData().

Implementation

Library call implemented in libmusic.a V21.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

ssplSetUserData()

ssplIsSpoolerActive

Tests if spooler has anything in its active queue.

Synopsis

```
bool ssplIsSpoolerActive (const SoundSpooler *sspl)
```

Description

Returns TRUE if the SoundSpooler has any SoundBufferNodes in its active queue. Simply calls `ssplGetSpoolerStatus()` and tests the `SSPL_STATUS_F_ACTIVE` flag.

Arguments

`sspl`
Pointer to a SoundSpooler structure.

Return Value

TRUE if SoundSpooler has any SoundBufferNodes in its active queue, FALSE otherwise.

Caveats

This state of `SSPL_STATUS_F_ACTIVE` is undefined when inside any SoundSpooler callback function. This is the case because the callback functions can be called inside the loop that processes the active queue. No guarantee is made about whether a SoundBufferNode is removed from the active queue before or after the callback functions are called.

Implementation

Macro implemented in `<:audio:soundspooler.h>` V24.

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`

See Also

`ssplGetSpoolerStatus()`, `ssplSendBuffer()`, `ssplProcessSignals()`

ssplPause

Pauses the SoundSpooler.

Synopsis

```
Err ssplPause ( SoundSpooler *sspl )
```

Description

ssplPause() pauses SoundSpooler's sample player instrument. Stops playback and retains the sample data position so that playback can be resumed at the same position with ssplResume(). This differs from the ssplStartSpoolerTags() ssplStopSpooler() pair which do not retain the current sample playback position.

The playback cannot be resumed from the paused location if ssplStartSpoolerTags() or ssplStopSpooler() are called between ssplPause() and ssplResume(). Multiple calls to ssplPause() have no effect; pause does not nest.

Sets SSPL_STATUS_F_PAUSED.

Arguments

sspl
Pointer to a SoundSpooler structure.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplResume(), ssplStopSpooler(), ssplGetSpoolerStatus(),
PauseInstrument()

ssplPlayData

Waits for next available buffer then sends a block full of data to the spooler. (convenience function)

Synopsis

```
int32 ssplPlayData ( SoundSpooler *sspl, char *Data, int32 NumBytes
)
```

Description

ssplPlayData() is a convenience function that does the following operations:

- Requests an available SoundBufferNode. If none are available, waits for one to become available.
- Attaches sample data to the SoundBufferNode.
- Submits the SoundBufferNode to the active queue.

Arguments

sspl
Pointer to a SoundSpooler structure.

Data
Pointer to data buffer.

NumBytes
Number of bytes to send.

Return Value

Positive value on success indicating the signal to be sent when the buffer finishes playing, or negative 3DO error code on failure. Never returns zero.

Implementation

Convenience call implemented in libmusic.a V21.

Notes

This function calls ssplProcessSignals() when forced to wait for a SoundBufferNode, but doesn't have a mechanism for the client to supply a UserBufferProcessor callback function. If you use this function and require completion notification, use a SoundBufferFunc instead of a UserBufferProcessor.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplSpoolData(), ssplRequestBuffer(), ssplSendBuffer(),
ssplSetBufferAddressLength(), ssplSetUserData(), ssplProcessSignals(),
SoundBufferFunc

ssplProcessSignals

Processes completion signals that have been received.

Synopsis

```
int32 ssplProcessSignals (  
    SoundSpooler *sspl, int32 SignalMask,  
    void (*UserBufferProcessor) (SoundSpooler *, SoundBufferNode *)  
)
```

Description

`ssplProcessSignals()` informs the spooler about which buffers have finished, so the spooler can move them to the free buffer queue.

If a `SoundBufferFunc` is installed, a `SSPL_SBMSG_COMPLETE` message is sent to it with each completed `SoundBufferNode` removed from the active queue. Also, a `SSPL_SBMSG_LINK_START` message is sent with the next buffer in the active queue. If `UserBufferProcessor` is non-NULL, each completed `SoundBufferNode` removed from the active queue is passed to it. If both a `SoundBufferFunc` and `UserBufferProcessor` are supplied, `ssplProcessSignals()` fails and returns `ML_ERR_BAD_ARG`.

If `SoundBufferFunc` fails, `ssplProcessSignals()` fails immediately and returns the error code returned by `SoundBufferFunc` without processing the rest of the signals. Calling `ssplProcessSignals()` again with the same signal set picks up where it left off.

Clears `SSPL_STATUS_F_ACTIVE` when all of the active buffers have completed (i.e. the flag is cleared when this function has processed the completion signals for all of the buffers that were in the active queue at the time that this function was called). You can use this to detect when the last submitted buffer has finished.

Arguments

`sspl`

Pointer to a `SoundSpooler` structure.

`SignalMask`

Signals received from `WaitSignal()`.

`UserBufferProcessor`

Function for each completed `SoundBufferNode`, or NULL.

Return Value

Non-negative value on success indicating number of completed buffers removed from the active queue, or negative 3DO error code on failure.

Implementation

Library call implemented in `libmusic.a V21`.

Notes

`ssplProcessSignals()` synchronizes the sound spooler's queues to the set of signals collected at the last `WaitSignal()`. Many of the other sound spooler functions assume that the sound spooler has been synchronized in this way (in particular `ssplSendBuffer()`). Do not call any sound spooler functions between a `WaitSignal()` involving sound spooler signals and `ssplProcessSignals()`.

Caveats

In the case where `SoundBufferFunc(SSPL_SBMSG_COMPLETE)` returns an error code, `SoundBufferFunc(SSPL_SBMSG_LINK_START)` isn't called for the next buffer in the active queue, because processing terminates immediately on receipt of the failure of `SoundBufferFunc(SSPL_SBMSG_COMPLETE)` even when `ssplProcessSignals()` is called again.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

`ssplSendBuffer()`, `ssplAbort()`, `SoundBufferFunc`, `UserBufferProcessor`, `ssplGetSpoolerStatus()`

ssplRequestBuffer

Asks for an available buffer.

Synopsis

```
SoundBufferNode *ssplRequestBuffer ( SoundSpooler *sspl )
```

Description

`ssplRequestBuffer()` returns an available `SoundBufferNode` from the free queue or `NULL` if none are available (all are in the active queue). When an available `SoundBufferNode` is returned, the client can attach sample data to it (by calling `ssplSetBufferAddressLength()`) and submit it to the active queue (by calling `ssplSendBuffer()`). This procedure is automatically done for you by the convenience functions `ssplSpoolData()` and `ssplPlayData()`.

This function gives you write permission to the returned `SoundBufferNode`. You have that write permission until the `SoundBufferNode` is returned to the `SoundSpooler` with `ssplSendBuffer()` or `ssplUnrequestBuffer()`.

Arguments

`sspl`
Pointer to a `SoundSpooler` structure.

Return Value

Pointer to a `SoundBufferNode` if one is available, or `NULL` if no buffers are available.

Implementation

Library call implemented in `libmusic.a V21`.

Examples

The following code fragment shows proper use of `ssplRequestBuffer()`, `ssplUnrequestBuffer()`, `ssplSetBufferAddressLength()`, and `ssplSendBuffer()`.

```
Err sendbuffer (SoundSpooler *sspl)
{
    SoundBufferNode *sbn;
    Err errcode;

    // request a buffer
    if ((sbn = ssplRequestBuffer (sspl)) != NULL) {

        // fill buffer, on failure return unused buffer
        // to free queue
        if ((errcode = fillbuffer (sspl, sbn)) < 0) {
            ssplUnrequestBuffer (sspl, sbn);
            return errcode;
        }

        // on success, send buffer (always moves buffer to
        // either free or active queue - no need to unrequest
        it)
        if ((errcode = ssplSendBuffer (sspl, sbn)) < 0) return
        errcode;
    }
}
```

```
        return 0;
    }

    Err fillbuffer (SoundSpooler *sspl, SoundBufferNode *sbn)
    {
        void *addr;
        uint32 len;

        // your code here to fill buffer - results in addr, len

        return ssplSetBufferAddressLength (sspl, sbn, addr, len);
    }
```

Notes

Prior to libmusic.a V24, SoundBufferNodes were actually `_removed_` from the free list; all knowledge of them was lost by the sound spooler until they were passed back to `ssplSendBuffer()` to put them back into one of the SoundSpooler's queues. If lost, `ssplDeleteSoundSpooler()` would lose memory and be unable to free some items.

Starting with V24, the sound spooler automatically tracks all sound buffer nodes requested by `ssplRequestBuffer()` in a "requested" queue. Any buffers that are not submitted or "unrequested", get put back into the free list by `ssplReset()` and freed by `ssplDeleteSoundSpooler()`.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

`ssplUnrequestBuffer()`, `ssplSendBuffer()`, `ssplSetBufferAddressLength()`,
`ssplSpoolData()`, `ssplPlayData()`

ssplReset

Resets SoundSpooler.

Synopsis

```
Err ssplReset (  
    SoundSpooler *sspl,  
    void (*UserBufferProcessor) (SoundSpooler *, SoundBufferNode *)  
)
```

Description

This function removes all SoundBufferNodes from the active queue and clears their associated signal bits. `ssplAbort()` calls this function to clear the active queue after stopping the sound spooler.

If a SoundBufferFunc is installed, a SSPL_SBMSG_ABORT message is sent to it with each SoundBufferNode removed from the active queue. If UserBufferProcessor is non-NULL, each SoundBufferNode removed from the active queue is passed to it. If both a SoundBufferFunc and UserBufferProcessor are supplied, `ssplReset()` fails and returns ML_ERR_BAD_ARG.

If SoundBufferFunc fails, `ssplReset()` fails immediately and returns the error code returned by SoundBufferFunc without removing the rest of the buffers from the active queue. Calling `ssplReset()` again safely picks up where it left off.

Clears SSPL_STATUS_F_ACTIVE.

Arguments

sspl

SoundSpooler to reset.

UserBufferProcessor

Pointer to a function to call with each SoundBufferNode removed from the active queue, or NULL.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Note

This function does not disturb the sample player instrument. If the spooler has not been stopped, any data that has already been queued up will continue to play, even though the associated SoundBufferNodes have been removed from the active queue. It is generally a good idea to call this function only after stopping the sound spooler, or use `ssplAbort()` instead.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

`ssplAbort()`, `SoundBufferFunc`, `UserBufferProcessor`,
`ssplGetSpoolerStatus()`

ssplResume

Resume SoundSpooler playback.

Synopsis

```
Err ssplResume ( SoundSpooler *sspl )
```

Description

`ssplResume()` resumes sound playback after being paused with `ssplPause()`. Playback is resumed from the sample data position in the spooled sound at which it was paused.

This function has no effect if `ssplStartSpoolerTags()` or `ssplStopSpooler()` are called between `ssplPause()` and `ssplResume()`. Multiple calls to `ssplResume()` have no effect; pause does not nest.

Clears `SSPL_STATUS_F_PAUSED`.

Arguments

`sspl`
Pointer to a SoundSpooler structure.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in `libmusic.a V21`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ssplPause()`, `ssplGetSpoolerStatus()`, `ResumeInstrument()`

ssplSendBuffer

Sends a buffer full of data.

Synopsis

```
int32 ssplSendBuffer ( SoundSpooler *sspl, SoundBufferNode *sbn )
```

Description

`ssplSendBuffer()` submits a `SoundBufferNode` ready for playback to the active queue. If the spooler has already been started, playback of this buffer will begin immediately if there are no other buffers in the active queue. In this case if a `SoundBufferFunc` is installed, a `SSPL_SBMSG_STARVATION_START` message is sent with `sbn`.

The `SoundBufferNode` is posted into the active buffer queue on success, or the free buffer list on failure. In either case, you are not permitted to modify the `SoundBufferNode` passed to this function after calling it.

When successful, sets `SSPL_STATUS_F_ACTIVE`.

Arguments

`sspl`

Pointer to a `SoundSpooler` structure.

`sbn`

Pointer to a `SoundBufferNode` to send (returned from `ssplRequestBuffer()`).

Return Value

Positive value on success indicating the signal to be sent when the buffer finishes playing, or negative 3DO error code on failure. Never returns zero.

Implementation

Library call implemented in `libmusic.a V21`.

Caveats

In the case where `ssplStartSpoolerTags()` is called with an empty active queue and `ssplSendBuffer()` actually starts the spooler, `ssplSendBuffer()` calls `SoundBufferFunc(SSPL_SBMSG_STARVATION_START)` instead of `SoundBufferFunc(SSPL_SBMSG_INITIAL_START)`. This isn't truly a starvation case, it's merely a difference in order of calls, but `ssplSendBuffer()` can't tell the difference.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ssplRequestBuffer()`, `ssplUnrequestBuffer()`,
`ssplSetBufferAddressLength()`, `ssplSpoolData()`, `ssplGetSpoolerStatus()`

ssplSetBufferAddressLength

Attaches sample data to a SoundBufferNode.

Synopsis

```
Err ssplSetBufferAddressLength ( SoundSpooler *sspl,  
                                SoundBufferNode *sbn,  
                                char *Data,  
                                int32 NumBytes )
```

Description

ssplSetBufferAddressLength() attaches sample data to a SoundBufferNode returned by ssplRequestBuffer(). This procedure is used internally by ssplPlayData() and ssplSpoolData().

Arguments

sspl
 Pointer to SoundSpooler that owns sbn.

sbn
 Pointer to a SoundBufferNode returned by ssplRequestBuffer().

Data
 Pointer to sample data to attach to this SoundBufferNode.

NumBytes
 Length of sample data (in bytes).

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Notes

The safe minimum size for spooler buffers appears to be around 256 bytes. Values less than this risk having gaps in the playback due insufficient time to process transitions from one sample Attachment to another. This minimum is in reality more a function of the time required to play a sample Attachment than that of sample length in bytes. The value 256 was determined using sampler_16_f2.dsp which at the time of this writing had the fastest DMA consumption of all the DSP instruments (176,400 bytes/sec). This means that this value should be a suitable minimum spooler buffer length for any sample playback instrument used with the sound spooler.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplRequestBuffer(), ssplSendBuffer(), ssplSpoolData()

ssplSetSoundBufferFunc

Install new SoundBufferFunc in SoundSpooler.

Synopsis

```
Err ssplSetSoundBufferFunc ( SoundSpooler *sspl, SoundBufferFunc  
func )
```

Description

Installs a new SoundBufferFunc callback function in a SoundSpooler.

Arguments

sspl

SoundSpooler to affect.

func

New callback function pointer to install, or NULL to disable callback.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V22.

Associated Files

<:audio:soundspooler.h>, libmusic.a

See Also

SoundBufferFunc

ssplSetUserData

Stores user data in a SoundBufferNode.

Synopsis

```
void ssplSetUserData ( SoundSpooler *sspl, SoundBufferNode *sbn,  
                      void *UserData )
```

Description

`ssplSetUserData()` stores user data pointer in a `SoundBufferNode`. This can be used to attach any extra data of your choosing to a `SoundBufferNode`. The user data pointer can be retrieved from a `SoundBufferNode` by calling `ssplGetUserData()`.

Arguments

`sspl`

Pointer to a `SoundSpooler` structure.

`sbn`

Pointer to a sound buffer node.

`UserData`

User data pointer to store in `SoundBufferNode`. Can be `NULL`.

Implementation

Library call implemented in `libmusic.a V21`.

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`

See Also

`ssplGetUserData()`, `ssplSpoolData()`

ssplSpoolData

Sends a block full of data. (convenience function)

Synopsis

```
int32 ssplSpoolData ( SoundSpooler *sspl, char *Data,  
                    int32 NumBytes, void *UserData )
```

Description

ssplSpoolData() is a convenience function that does the following operations:

- Requests an available SoundBufferNode, returns 0 if none are available.
- Attaches sample data and optional user data pointer to the SoundBufferNode.
- Submits the SoundBufferNode to the active queue.

Arguments

sspl
Pointer to a SoundSpooler structure.

Data
Pointer to the buffer of data.

NumBytes
Number of bytes of data to send.

UserData
Pointer to user data, or NULL.

Return Value

Positive value on success indicating the signal to be sent when the buffer finishes playing, zero if no SoundBufferNodes are available, or negative 3DO error code for others kinds of failure.

Implementation

Convenience call implemented in libmusic.a V21.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplPlayData(), ssplRequestBuffer(), ssplSendBuffer(),
ssplSetBufferAddressLength(), ssplSetUserData()

ssplStartSpoolerTags

Starts SoundSpooler.

Synopsis

```
Err ssplStartSpoolerTags ( SoundSpooler *sspl, const TagArg
                          *startTagList )

Err ssplStartSpoolerTagsVA ( SoundSpooler *sspl, uint32 startTag1,
                          ... )
```

Description

ssplStartSpoolerTags() starts the sound spooler's sample player instrument. It starts playback at the beginning of the first buffer in the active queue (submitted by ssplSendBuffer()). If there are no buffers in the active queue, playback will not begin until a buffer is submitted with a call to ssplSendBuffer().

If there is a SoundBufferFunc installed and there is a buffer in the active queue, sends a SSPL_SBMSG_INITIAL_START message with a pointer to the first active buffer. If SoundBufferFunc returns an error code, ssplStartSpoolerTags() terminates and returns that error code.

This function sets SSPL_STATUS_F_STARTED and clears SSPL_STATUS_F_PAUSED.

Arguments

sspl
Pointer to a SoundSpooler structure.

startTagList
Tag list be passed to StartInstrument() for sample player instrument. Useful tags include AF_TAG_AMPLITUDE_FP and AF_TAG_SAMPLE_RATE_FP. See StartInstrument() for details.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in libmusic.a V21.

Module Open Requirements

OpenAudioFolio()

Associated Files

<:audio:soundspooler.h>, libmusic.a, System.m2/Modules/audio

See Also

ssplStopSpooler(), ssplPause(), ssplSendBuffer(), SoundBufferFunc, ssplGetSpoolerStatus(), StartInstrument()

ssplStopSpooler

Stops SoundSpooler.

Synopsis

```
Err ssplStopSpooler ( SoundSpooler *sspl )
```

Description

`ssplStopSpooler()` stops the SoundSpooler's sample player instrument. Unlike `ssplPause()`, this call does not retain the current sample data position. Starting the spooler after stopping it will start sound playback from the beginning of the first buffer in the active queue.

This function merely stops the SoundSpooler's sample player instrument. It does not disturb the active queue.

This function clears `SSPL_STATUS_F_STARTED` and `SSPL_STATUS_F_PAUSED`.

Argument

`sspl`
Pointer to a SoundSpooler structure.

Return Value

Non-negative value on success, or negative 3DO error code on failure.

Implementation

Library call implemented in `libmusic.a V21`.

Module Open Requirements

`OpenAudioFolio()`

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`, `System.m2/Modules/audio`

See Also

`ssplStartSpoolerTags()`, `ssplPause()`, `ssplAbort()`,
`ssplGetSpoolerStatus()`, `StopInstrument()`

ssplUnrequestBuffer

Returns an unused buffer to the sound spooler.

Synopsis

```
Err ssplUnrequestBuffer (SoundSpooler *sspl, SoundBufferNode *sbn)
```

Description

`ssplUnrequestBuffer()` puts an unused `SoundBufferNode` from the requested queue back into the free queue. Use it as a way to politely give up write access to a `SoundBufferNode` that was given to you by `ssplRequestBuffer()` that you aren't going to submit to `ssplSendBuffer()`. For example, This can be used as part of a clean up path that traps failures between `ssplRequestBuffer()` and `ssplSendBuffer()`.

This is automatically done for all `SoundBufferNodes` in the requested list by `ssplReset()`, `ssplAbort()`, and `ssplDeleteSoundSpooler()`.

Arguments

`sspl`

Pointer to a `SoundSpooler` structure.

`sbn`

Pointer to a `SoundBufferNode` to return. The SBN must actually be in the requested list or else this function will return an error.

Return Value

Non-negative value on success, negative error code on failure:

`ML_ERR_BAD_ARG`

The supplied `sbn` was not actually on the requested list.

Implementation

Library call implemented in `libmusic.a V24`.

Associated Files

`<:audio:soundspooler.h>`, `libmusic.a`

See Also

`ssplRequestBuffer()`, `ssplSendBuffer()`

UserBufferProcessor

Callback function prototype called by `ssplProcessSignals()`, `ssplAbort()`, and `ssplReset()`.

Synopsis

```
void (*UserBufferProcessor) ( SoundSpooler *sspl, SoundBufferNode
*sbn )
```

Description

This user callback function is called by `ssplProcessSignals()` for each buffer that has finished playing and `ssplAbort()` for each buffer that is removed from the active buffer queue. This function can be used as notification that the sound spooler is done with a given buffer. For example, if sound is being spooled from a network connection, this function could be used to reply the network packet that contained the sample data for a given buffer.

Using the `SoundBufferNode`'s `UserData` field can help identify the buffer being passed to this function (for example, `UserData` could be a pointer to the aforementioned network packet). See `ssplGetUserData()`.

It is not legal to do anything to disturb the sound spooler's active buffer queue during this function. In particular, do not call `ssplSendBuffer()` or anything that might call `ssplSendBuffer()` as this will confuse the list processor in `ssplProcessSignals()`.

The state of `SSPL_STATUS_F_ACTIVE` is undefined when inside a `UserBufferProcessor` function.

Arguments

`sspl`
Pointer to the `SoundSpooler` passed to `ssplProcessSignals()` or `ssplAbort()`.

`sbn`
Pointer to the completed (or aborted) `SoundBufferNode` that was just removed from the active queue.

Implementation

Callback function prototype used by `libmusic.a V22`.

Note

This facility has been superseded by the `SoundBufferFunc` system.

There are plenty of sound spooler functions in which buffers can be removed from the active queue without any means for the client to provide a `UserBufferProcessor` function. This list includes, but is not limited to `ssplDeleteSoundSpooler()`, `ssplAttachInstrument()`, `ssplDetachInstrument()`, and `ssplPlayData()`. Consider using a `SoundBufferFunc` instead of a `UserBufferProcessor`.

See Also

`SoundBufferFunc`, `ssplProcessSignals()`, `ssplAbort()`, `ssplReset()`

